AFRL-IF-RS-TR-2005-67
Final Technical Report
March 2005

DYNAMIC POLICY EVALUATION FOR CONTAINING NETWORK ATTACKS (DEFCN)

University of Southern California at Marina Del Rey

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K293

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

REPORT DOCUMENTATION PAGE				*Form Approved* *OMB No. 074-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** MARCH 2005	**3. REPORT TYPE AND DATES COVERED** Final Jul 00 – Dec 03		
4. TITLE AND SUBTITLE DYNAMIC POLICY EVALUATION FOR CONTAINING NETWORK ATTACKS (DEFCN)				**5. FUNDING NUMBERS** C - F30602-00-2-0595 PE - 62301E PR - K293 TA - 33 WU - A1
6. AUTHOR(S) B. Clifford Neuman, Dongho Kim and Tatyana Ryutov				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Southern California at Marina del Rey Information Sciences Institute 4676 Admiralty Way Suite 1001 Marina del Rey California 90292				**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington Virginia 22203-1714 / AFRL/IFGA 525 Brooks Road Rome New York 13441-4505				**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-2005-67
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Jon B. Valente/IFGA/(315) 330-1559/ Jon.Valente@rl.af.mil				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				**12b. DISTRIBUTION CODE**
13. ABSTRACT *(Maximum 200 Words)*				

The DARPA funded DEFCN project at USC's Information Sciences Institute has developed an access control framework that is sensitive to network threat conditions. Project members developed the Generic Authorization and Access control Application Programming Interface (GAA-API), a middle-ware API for generic authorization and access-control and have integrated this framework with intrusion detection and response systems. Access policies evaluated by the GAA-API can be conditioned upon network threat conditions communicated by intrusion detection systems, and they also adapt to changes in information sharing policies prompted by the formation of dynamic coalitions. The GAA-API allows the generation of audit records at the control points in applications. The level of detail of the audit records generated is dependent upon the network threat condition and on authentication characteristics of a request.

14. SUBJECT TERMS Dynamic Policy Management, Authorization, Access Control, Intrusion Response, Dynamic Coalitions			**15. NUMBER OF PAGES** 84 / **16. PRICE CODE**
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 **Standard Form 298 (Rev. 2-89)**
Prescribed by ANSI Std. Z39-18
298-102

Table of Contents

1. Introduction

The DARPA funded DEFCN project at USC's Information Sciences Institute has developed an access control framework that is sensitive to network threat conditions. Project members developed the Generic Authorization and Access control Application Programming Interface (GAA-API), a middle-ware API for generic authorization and access-control, and have integrated this framework with intrusion detection and response systems. Access policies evaluated by the GAA-API can be conditioned upon network threat conditions communicated by intrusion detection systems, and they also adapt to changes in information sharing policies prompted by the formation of dynamic coalitions. The GAA-API allows the generation of audit records at the control points in applications. The level of detail of the audit records generated is dependent upon the network threat condition and on the authentication characteristics of a request.

Some security countermeasures are too costly to apply all the time. A system's security and access control mechanisms must adapt to varying threat conditions in order to adequately respond to attacks while imposing minimal hindrance to legitimate users. For example, when a threat is detected, servers should modify logging behavior to record additional detail and forward the detail to a central server for analysis. In addition, one might want to adapt policies in a crisis situation to allow greater sharing of data, but in a controlled manner. This way, coalition members from different agencies or governments can share data more readily, once a coalition is formed, without having to "turn off" security as might otherwise happen if sharing was needed and turning off security was the only way to allow it. Other countermeasures, like market-based methods for resource allocation, should be enabled when needed to thwart denial of service attacks but disabled during normal operation.

Higher level software must adapt its behavior in response to a perceived threat in order to enable fine-grained response to an attack scenario, while ensuring that responses are comprehensive enough to protect against attacks that originate from multiple points. This support must be placed in servers, at the application level, because often only the servers themselves have knowledge of the operations that are requested or of the objects to be manipulated by the request. ISI's research focused on the application level response to attacks, and provided a mechanism through which application responses were coordinated on a system-wide basis.

For a system-wide response to occur across servers, the servers must implement an access control system that can adapt to threat conditions. Because this kind of adaptation is not of particular concern to application developers, who are rightfully more concerned with the routine operation of their servers, these features were embedded within an access control system that provided benefit to the application developer.

The technical part of this report is structured as follows. The introduction provided a high-level overview of the goals and accomplishment of the DEFCN effort. Section 2 describes the products and development activities that were performed under the project. Section 3 describes the activities of the effort by quarter. Brief reports of the meetings and conferences attended are included chronologically in section 4.

Greater details regarding the results of the DEFCN effort are found in the publications that are listed in sections 5 and 6 and appear in their entirety in the appendix. In particular, the paper "Dynamic Authorization and Intrusion Response in Distributed Systems," from the Proceedings of the 3rd DARPA Information Survivability Conference and Exposition, provides a good overview of the results of the DEFCN effort.

Section 7 lists the personel associated with the project and degrees awarded to students whose work was funded under the DEFCN effort.

2. Product and Development Activities

Greater detail about the development activities described in this section may be found in section 3, and in the papers listed in sections 5 and 6, with copies of the papers in appendix A. The latest code distributions resulting from the DEFCN effort may be dowloaded from:

http://gost.isi.edu/projects/defcn/ or http://defcn.sysproject.info

The software developed and extended by the Global Operating Systems Technologies (GOST) project includes the following:

2.1. GAA-API (Generic Authorization & Access-control Application Programming Interface)

Applications invoke GAA-API functions to check authorization against an authorization model. The API functions obtain policies from local files, distributed authorization servers, and from credentials provided by the user. The functions combine local and distributed authorization information under a single API based on the requirements of the application and applicable policies, and request credentials if necessary.

A module is provided to support a simple policy language called Extended Access Control Lists (EACL) that is designed to describe user-level authorization policy. The GAA-API provides a general-purpose execution environment in which EACLs are evaluated. EACL is one of the possible policy languages supported by the GAA-API. The underlying architecture of the GAA-API allows different policy languages to co-exist in one execution environment.

2.2. GAA-API integrated Apache web server

The GAA-API was integrated with Apache web server versions 1.3 and 2.0.4. Integration with the GAA-API provided the following additional capabilities beyond those provided by the Apache without GAA-API integration:

1. **Flexible, adaptive and fine grained access control** - Apache supports only limited identity and host based policies that deny/allow access to protected resources. The policies supported by the GAA-API allow security administrators to control not only which users or groups and from which locations are allowed access, but also support other conditions, including time, location, system load, and system threat level. Furthermore, the GAA-Apache supports adaptive security policies, which respond to attacks by modifying security measures automatically. For example, policies can be specified that enable restricting access to local users only or requesting extra credentials.

2. **Fine grained audit and notification capabilities** - Besides making decisions of whether a request is accepted or rejected, the GAA-API libraries provide routines that can execute certain actions, such as logging information, notifying the administrator, etc. Furthermore, the GAA-API supports fine-tuning of the notification and audit services that helps to increase the quality and reduce the volume of generated data and alerts.

3. **Application level intrusion/misuse detection and response capability** - Apache access control mechanisms were not designed to aid the detection of threats or to adjust their behavior based on perceived threat conditions. The GAA-Apache supports for policies that assist in detecting and responding to application level intrusions and adapt to perceived system threat conditions. For example, the system can reject or modify the requests that violate security policy (e.g., request encapsulates dangerous characters).

4. **Flexible policy composition framework that relates separately defined policies** - The GAA-Apache supports system wide and local policies. The composed policy is constructed by merging the system wide and local policies. This separation of policies allows for flexible and efficient policy management and also enables coordination of policy across multiple applications.

2.3. GAA-API integrated SSH (Secure Shell server)

The GAA-API was integrated with the SSH server. The GAA-API reads policy information from the target users ".ssh" directory and applies those policies to determine whether remote login is allowed to a particular account. The changes made to integrate the GAA-API with SSH has been included in the GAA reference implementation illustrative application on how to use the GAA-API from an application.

2.4. GAA-API integrated IPSec (FreeSWAN)

The GAA-API was integrated with the FreeS/WAN IPsec version 1.91. Integration with the GAA-API provided the following additional capabilities beyond those provided by the IPsec without GAA-API integration:

- replaced hard coded parameter selection for the Security Association (SA) negotiation with fine-grained controls over the parameters;

- added time and location based controls;

- added connection duration controls.

It also enables the GAA-IPsec to support dynamic policies that adapt to current system threat condition. For example, policies can be specified to:

- require stronger authentication and/or encryption methods (SA parameters) when system is under attack;

- deny connections to less trusted IP addresses;

- control the number and duration of the IPsec connection when the system is under ongoing or suspected DoS attack.

2.5. GAA-API integrated SOCKS5 (proxy server)

The GAA-API was integrated with the Socks5 version 1.0r11 Integration with the GAA-API provided the following additional capability beyond the one provided by the application without GAA-API integration: checking for source and destination IP addresses and other conditions such as time, system threat level, secondary proxy IP adddress, content of URL, and authentication information. The GAA-API can also inspect the request frequency to limit the consumption of each user and protect the system from potential DoS attacks. In addition, the GAA-API can also be used to perform dynamic logging, notification, and the coodination between other applications such as firewall. These changes enable coordination of policy with other applications. It also enables the SOCKS 5 to support dynamic policies that adapt to system threat level. For example, policies can be specified to disable or block outgoing traffic as the system threat level escalates.

2.6. Trust Builder integrated GAA-API

The GAA-API was integrated with the TrustBuilder system. The GAA-API neither supports trust negotiation nor protection of sensitive policies. The TrustBuilder regulates when and how sensitive information is disclosed to other parties; however, the system lacks fine-grained adaptive policies. This combination extends the capabilities of each system. In particular, the GAA-API/TrustBuilder integration allows us to do the following:

- detect and thwart certain attacks on electronic business transactions (e.g., some types of DoS and sensitive information leaks);

- support cost effective trust negotiation, such that TrustBuilder is invoked only when negotiation is required by access control policies;

- dynamically adapt information disclosure and resource access policies according suspicion level and general system threat level.

3. Publications in Technical Journals

- "Integrated Access Control and Intrusion Detection for Web Servers" Tatyana Ryutov, Clifford Neuman, Dongho Kim and Li Zhou. IEEE Transactions on Parallel and Distributed Systems, Vol. 14, No. 9, September 2003.

4. Papers Presented at Meetings, Conferences, Seminars, etc.

- "Integrated Access Control and Intrusion Detection for Web Servers," Tatyana Ryutov, Clifford Neuman, Dongho Kim and Li Zhou. In Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003), Providence, Rhode Island, May 2003.

- "Dynamic Authorization and Intrusion Response in Distributed Systems," Tatyana Ryutov, Clifford Neuman, and Dongho Kim. In Proceedings of the 3rd DARPA Information Survivability Conference and Exposition (DISCEX III), Washington, D.C. April 22-24, 2003.

- "The Specification and Enforcement of Advanced Security Policies," Tatyana Ryutov, Clifford Neuman. In Proceedings of the Conference on Policies for Distributed Systems and Networks (POLICY 2002), June 5-7, 2002, in Monterey, California.

- "The Set and Function Approach to Modeling Authorization in Distributed Systems," Tatyana Ryutov, Clifford Neuman. In Proceedings of the Workshop on Mathematical Methods and Models and Architecture for Computer Networks Security, May 2001, St. Petersburg, Russia.

- "Representation and Evaluation of Security Policies for Distributed System Services," Tatyana Ryutov, Clifford Neuman. In Proceedings of the DARPA Information Survivability Conference & Exposition, January 2000. Hilton Head, South Carolina.

5. Personnel

- B. Clifford Neuman - Senior Research Scientist and Associate Division Director
- Ho Suk Chung - Graduate Research Assistant
- Dongho Kim - Computer Scientist
  Ph.D. in Computer Science, University of Southern California, 2001
  Thesis: Reconstructing Interconnections on Disconnected Mobile Hosts
- Tatyana Ryutov - Computer Scientist
  Ph.D. in Computer Science, University of Southern California, August 2002
  Thesis: Control-driven Authorization Model for Distributed System Services
- Li Zhou - Graduate Research Assistant
- Arnold Diaz - Project Support

6. New Discoveries, inventions, or patent disclosures

No patents were filed for as the result of the research in the DEFCN effort. In the interest of promoting the dissemination of the important ideas resulting from this research, results were published and such prior publication precludes the patenting of such results.

APPENDIX
Publications and Papers Presented at Meetings and Conferences

For more information on the results of the DEFCN effort, refer to the publications listed below. Note that the paper entitled "Dynamic Authorization and Intrusion Response in Distributed Systems," from the Proceedings of the 3rd DARPA Information Survivability Conference and Exposition, provides a good overview. Publications in their entirety will follow this page.

- "Integrated Access Control and Intrusion Detection for Web Servers" Tatyana Ryutov, Clifford Neuman, Dongho Kim and Li Zhou. IEEE Transactions on Parallel and Distributed Systems, Vol. 14, No. 9, September 2003.

- "Integrated Access Control and Intrusion Detection for Web Servers," Tatyana Ryutov, Clifford Neuman, Dongho Kim and Li Zhou. In Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003), Providence, Rhode Island, May 2003.

- "Dynamic Authorization and Intrusion Response in Distributed Systems," Tatyana Ryutov, Clifford Neuman, and Dongho Kim. In Proceedings of the 3rd DARPA Information Survivability Conference and Exposition (DISCEX III), Washington, D.C. April 22-24, 2003.

- "The Specification and Enforcement of Advanced Security Policies," Tatyana Ryutov, Clifford Neuman. In Proceedings of the Conference on Policies for Distributed Systems and Networks (POLICY 2002), June 5-7, 2002, in Monterey, California.

- "The Set and Function Approach to Modeling Authorization in Distributed Systems," Tatyana Ryutov, Clifford Neuman. In Proceedings of the Workshop on Mathematical Methods and Models and Architecture for Computer Networks Security, May 2001, St. Petersburg, Russia.

- "Representation and Evaluation of Security Policies for Distributed System Services," Tatyana Ryutov, Clifford Neuman. In Proceedings of the DARPA Information Survivability Conference & Exposition, January 2000. Hilton Head, South Carolina.

Appendix A:

Integrated Access Control and Intrusion Detection for Web Servers

Tatyana Ryutov, Clifford Neuman, *Senior Member*, *IEEE*, Dongho Kim, *Member*, *IEEE*, and Li Zhou

Abstract—Current intrusion detection systems work in isolation from access control for the application the systems aim to protect. The lack of coordination and interoperation between these components prevents detecting and responding to ongoing attacks in real-time before they cause damage. To address this, we apply dynamic authorization techniques to support fine-grained access control and application level intrusion detection and response capabilities. This paper describes our experience with integration of the Generic Authorization and Access Control API (GAA-API) to provide dynamic intrusion detection and response for the Apache Web server. The GAA-API is a generic interface which may be used to enable such dynamic authorization and intrusion response capabilities for many applications.

Index Terms—Access control, authorization, security policy, intrusion detection, Apache Web server.

1 Introduction and Motivation

Web servers continue to be attractive targets for attackers seeking to steal or destroy data, deny user access, or embarrass organizations by changing Web site contents. The Web servers are an easy target for outside intruders because the servers must be publicly available around the clock. In order to penetrate their targets, attackers may exploit well-known service vulnerabilities. A Web server can be subverted through vulnerable CGI scripts, which may be exploited by metacharacters or buffer overflow attacks. These vulnerabilities may be related to the default installation of the server, or may be introduced by careless writing of custom scripts.

Web servers are also popular targets for Denial of Service (DoS) attacks. An attacker sends a stream of connection requests to a server in an attempt to crash or slow down the service. Launching a DoS attack against a Web server can be accomplished in many ways, including ill-formed HTTP requests (e.g., a large number of HTTP headers). As the server tries to process such requests, it slows down and becomes unable to process other requests. In addition, Web servers exhibit susceptibility to password guessing attacks.

To address these risks, Web servers require increased security protection. Effective system security starts with security policies that are supported by an access control mechanism. The access control policy to be enforced should depend on the current state of the system (e.g., time of day, system load, or system threat level). More restrictive organizational policies may be enforced after hours when the system is busy or if suspicious activity has been detected.

Unfortunately, many Web servers (e.g., Apache and IIS) support only limited identity and host-based policies that deny/allow access to protected resources. The policies are checked only when an access request is received to determine whether the request should be permitted or forbidden. These policies do not support observing and reporting suspicious activity (e.g., embedding hexadecimal characters in a query) and modifying system protection as a result.

Thus, the security policies must not only specify legitimate user privileges, but also aid in the detection of threats and adapt their behavior based on perceived system threat conditions. Even a single instance of a request for a vulnerable CGI script or malformed request should be reported immediately and countermeasures should be applied. Such countermeasures may include:

- **generating audit records;**
- **notifying network servers that are monitoring security relevant events in the system;**
- **tightening local policies** (e.g., restricting access to local users only or requesting extra credentials); and
- **modifying overall system protection**. Examples include terminating the session, logging the user off the system, disabling local account or blocking connections from particular parts of the network, or stopping selected services (e.g., disable ssh connections).

These actions would be followed by an alert to the security administrator, who can then assess the situation and take the appropriate corrective actions. This step is important since an automated response to attacks can be used by an intruder in order to stage a DoS attack (the intruder could have impersonated a host or a user).

Traditional access control mechanisms were not designed to aid the detection of threats or to adjust their behavior based on perceived threat conditions. Common countermeasures to Web server threats depend on separate components like firewalls, Intrusion Detection Systems (IDSs), and code integrity checkers. While these components are useful in detecting some kinds of attacks, they do not fully address a Web server's security needs. For example, firewalls can deny access to unauthorized network connections; however, they cannot stop attacks coming in via authorized ports. In the general case, IDSs

- *The authors are with USC Information Sciences Institute, 4676 Admiralty Way, Suite 1001, Marina del Rey, CA 90292.*
  E-mail: {tryutov, bcn, dongho, zhou}@isi.edu.

Manuscript received 24 Feb. 2003; accepted 15 May 2003.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 118642.

provide only incomplete coverage, leaving sophisticated attacks undetected. Other disadvantages include a large number of false positives and the inability to preemptively respond to attacks. Integrity checkers can detect unauthorized changes to files on a Web site, but only after the damage has been done.

Motivated by the multitude of Web server vulnerabilities and generally unsatisfactory server protection, we propose an integrated approach to Web server security—the Generic Authorization and Access-control API (GAA-API) that supports fine-grained access control and application level intrusion detection and response.

The API evaluates HTTP requests and determines whether the requests are allowed and if they represent a threat according to a policy. Our approach differs from other work done in this area by supporting access control policies extended with the capability to identify intrusions and respond to the intrusions in real-time. The policy enforcement takes three phases:

1. Before the requested operation (e.g., display an HTML file or run a CGI program) starts—to decide whether this operation is authorized.
2. During the execution of the authorized operation—to detect malicious behavior in real-time (e.g., a process consumes excessive system resources).
3. After the operation is completed—to activate post execution actions, such as logging and notification whether the operation succeeds or fails (e.g., alerting that a particular critical file was written can trigger a process to check the contents of the file).

By being integrated with the Web server and having the ability to control the three processing steps of the requested operation, the GAA-API can respond to suspected intrusion in real-time before it causes damage, whether it is site defacement, data theft, or a DoS attack.

A Web server has to be modified in order to utilize the GAA-API. However, once the relatively easy integration is completed, it becomes possible to handle access control decisions and application level intrusion detection simultaneously. Furthermore, since the GAA-API is a generic tool, it can be used by a number of different applications with no modifications to the API code. In this paper, we focus on the Web server. However, the API can provide enhanced security for applications with different security requirements. We have integrated the GAA-API with the Apache Web server, SOCKS5, sshd, and FreeS/WAN IPsec for Linux.

2 Policy Representation

The Extended Access Control List (EACL) is a simple language that we implemented to describe security policies that govern access to protected resources and identify threats that may occur within application and specify intrusion response actions [5]. An EACL is associated with an object to be protected. It specifies positive and negative access rights with an optional set of associated conditions that describe the context in which each access right is granted or denied.

An EACL describes more than one set of disjoint policies. The policy evaluation mechanism is extended with the ability to read and write system state. The implementation is based on conditions that provide support for monitoring and updating internal system structures and their runtime behaviors.

A condition may either explicitly list the value of a constraint or specify where the value can be obtained at runtime. The latter allows for adaptive constraint specification since allowable times, locations, and thresholds can change in the event of possible security attacks. The value of condition can be supplied by other services, e.g., an IDS. All conditions are classified as:

- **Preconditions** specify what must be true in order to grant the request (e.g., access identity, time, location, and system threat level).
- **Request-result** conditions must be activated whether the authorization request is granted or whether the request is denied (e.g., audit, notification, and threshold).
- **Midconditions** specify what must be true during the execution of the requested operation (e.g., a CPU usage threshold that must hold during the operation execution).
- **Postconditions** are used to activate post execution actions, such as logging and notification whether the operation succeeds or fails.

A **condition block** defines a conjunction of a totally ordered set of conditions. Conditions are evaluated in the order they appear within a condition block. An **EACL entry** consists of a positive or negative access right and four optional condition blocks: a set of preconditions, a set of request-result conditions, a set of midconditions, and a set of postconditions. An **EACL** consists of an ordered set of disjunctive EACL entries. An EACL representation supports disjunction and conjunction of conditions to activate different control modes. A transition between the disjoint EACL entries is regulated automatically by reading the system state (e.g., time of day or the system threat level).

In the current framework, the evaluation of entries within an EACL and evaluation of conditions within an EACL entry is totally ordered. Evaluation of an EACL starts from the first to the last in the list of EACL entries. The resolution of inconsistent authorizations is based on ordering. The entries which have already been examined take precedence over new entries. The order has to be assessed before EACL evaluation starts. Determining the evaluation order is currently done by a policy officer. We recognize that the function of defining the order of EACL entries and conditions within an entry can be best served by an automated tool to ensure policy correctness and consistency and to ease the policy specification burden on the policy officer. We plan to design and implement such tool in the future. The GAA-API provides a general-purpose execution environment in which EACLs are evaluated.

2.1 Policy Composition

Policy composition is a process of relating separately specified policies. Our framework supports system-wide and local policies. This separation is useful for efficient policy management. Instead of repeating policies that apply to all applications in individual application policies, we define these policies as a separate *system-wide policy* that is applied globally and is consulted on all the accesses to all applications. *Local policies* allow users and applications to

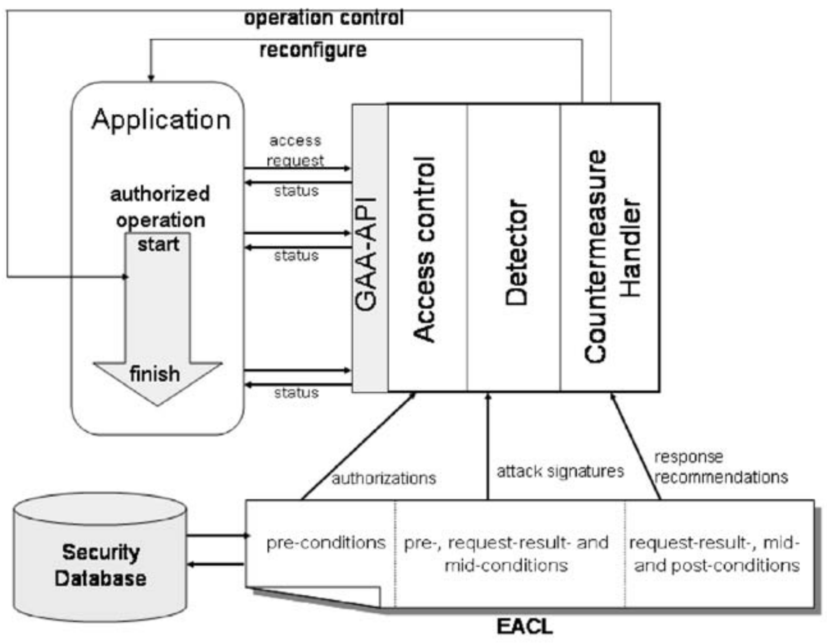

Fig. 1. Generic Intrusion Detection and response.

define their own policy in addition to the global one. The composed policy is constructed by merging the system-wide and local policies. First, system-wide policies are retrieved and placed at the beginning of the list of policies. Then, the local policies are retrieved and added to the list. Thus, system-wide policies implicitly have higher priority than the local policies.

A system-wide policy specifies a ***composition mode*** that describes how local policies are to be composed with the system-wide policy. The framework supports three composition modes:

- **Expand.** A system-wide policy broadens the access rights beyond those granted by local policies. It is the equivalent of a disjunction of the rights. The access is allowed if either the system-wide or the local policy allows the access. This is useful to ensure that a request permitted by the system-wide policy cannot fail due to access rejection at the local level.
- **Narrow.** A system-wide policy narrows the access rights so that objects cannot be accessed under particular conditions regardless of the local policies. The policy that controls access to an object may have mandatory and discretionary components. Generally, mandatory policy is set by the domain administrator, while discretionary policy is set by individuals or applications. The mandatory policies must always hold. The discretionary policies must be satisfied in addition to the mandatory policies. Thus, the resulting policy represents the conjunction of the mandatory and discretionary policies.
- **Stop.** If a system-wide policy exists, that policy is applied and local policies are ignored. An administrator may require complete overriding of the local policies with the system-wide policies. This is useful in order to react quickly to an attack. One might use the **stop** mode to shut down certain component systems. This is also useful when the administrator wants to, for example, allow access to a document only to himself. If he specifies a policy using the **expand** mode, then additional access can be granted at the local level. If he uses **narrow** mode, the local policies could add additional restrictions that can deny the access.

To evaluate several separately specified local (or system-wide) policies, we take a conjunction of the policies.

3 Generic Application Level Intrusion Detection Framework

The system detects intrusions by comparing access request patterns against the security policies and taking some actions if the request is judged to be suspicious. Because this applies to any application, this portion of the system can be fairly generic and used for a number of applications. However, the database of known intrusion scenarios, attack patterns, and responses should be customized for different applications. The customization is done through specification of policies expressed in EACL format. Fig. 1 shows a high-level view of our framework.

The **access control** module mediates access requests generated by applications and forwarded to the GAA-API for approval. The **detector** examines access requests and determines the presence of an attack based on the policies. If the detector determines the request to be suspicious, the **countermeasure handler** will take the corrective actions to prevent malicious actions from being executed. The **Security Database** provides information collected from various sources including: user activity, misuse signatures and intrusion scenarios, application audit records, etc. The **Security policy (EACL)** contains:

- Positive and negative authorizations checked by the access control module.

- The information to be analyzed by the detector (e.g., database of attacks, user activity profiles, parameters of an access request and information obtained from monitoring the execution of the requested operation, and a status (success or failure) of the completed operation).
- Actions to be performed by the countermeasure handler for incident response—the system may deny requested access, affect execution of the requested operation (e.g., suspend or kill a process), generate alarms and audit records, update firewall rules, and so on.

4 GAA-API and IDS Interactions

The data extracted from an application at the access control time can be supplemented with data from a network and host-based IDSs to detect attacks not visible at the application level and reduce the false alarm rate. The current GAA-API interaction with IDS is limited to determining the current system threat profile and adapting the security policy to respond to changing security requirements. Our next task is to support closer interaction between the GAA-API and different IDSs.

4.1 "GAA-API to IDS" Interactions

Here are the kinds of information that the GAA-API can report to an IDS:

1. **Ill-formed access requests**. Because the GAA-API processes access control requests by applications, the API can apply application-level knowledge to determine whether the request is properly formed. Ill-formed access requests may signal an attack. For example, consider an application that issues queries to a database. It is assumed that the application makes bug-free database queries. If there are errors in the access request, it may indicate that someone has compromised the application server and is performing ad hoc queries against the database.
2. **Accesses requests with abnormal parameters**. The API can report accesses requests with parameters that violate site policy or are abnormally large.
3. **Denied access**. The API can report even a single instance of access denial to sensitive system objects. The API can report attempts to access nonexistent hosts on a network, which could indicate network scanning or mapping activity and attempts to use critical commands.
4. **Exceeding thresholds**. Examples of types of events that can be controlled by the threshold detectors and reported by the GAA-API include the number of failed login attempts within a given period of time.
5. **Incidents**. The GAA-API can report detected application-level attacks.
6. **Suspicious application behavior**. The API can report unusual application behavior such as read only application creating files.
7. **Legitimate activity**. The GAA-API can communicate access request information to IDS. This information can be used to derive profiles that describe the typical behavior of users working with different applications. An automatically developed profile can be created by an IDS module that collects and processes the information about granted access rights over time and forms a statistically valid sample of user behavior that can be used for anomaly detection.

4.2 "IDS to GAA-API" Interactions

The GAA-API can request a network-based IDS to report, for example, indications of address spoofing. This information can be used in addition to the application-level attack signatures to further reduce the false positive rate and avoid DoS attacks. This is particularly important for applying proactive countermeasures, such as updating firewall rules and dropping connections.

The API can request information for adjusting policies, such as values for thresholds, times, and locations. When implementing a threshold detector, the obvious difficulty is choosing the threshold number and a time interval of the analysis for a particular event. The values may depend on many factors and can be determined by a host-based IDS and communicated to the GAA-API.

5 GAA-Apache Integration

5.1 Apache Access Control

Apache's access control system [6] provides a method for Web masters to allow or deny access to certain URL paths, files, or directories. Access can be controlled by requiring username and password information or by restricting the originating IP address of the client request.

Access control is usually confined to specific directories of the document tree. When processing client's request to access a document, Apache looks for an access control file called .htaccess in every directory of the path to the document.

Here is a sample .htaccess file:

```
Order Deny, Allow
Deny from All
Allow from 10.0.0.0/255.0.0.0
AuthType Basic
AuthUserFile /usr/local/apache2/.htpasswd-isi-staff
Require valid-user
Satisfy All
```

The "`Allow from 10.0.0.0/255.0.0.0`" allows connections only from hosts within the specified IP range. All other hosts will get a "Permission Denied" message. The "Require valid-user" requires that the user enter a username and password. These username/password pairs are stored in a separate file specified by the `AuthUserFile` directive.

After receiving an access request, the Apache core modules call the *check_dir_access* function in *mod_access* or the *authenticate_basic_user*, *check_user_access* routines in *mod_auth* to check access control policies. A structured parameter *request_rec* is provided to the routines, containing information about the request. Finally, every routine returns the decision to the core modules. Three output values are defined: `HTTP_OK`—the request is granted; `HTTP_DECLINED`—the

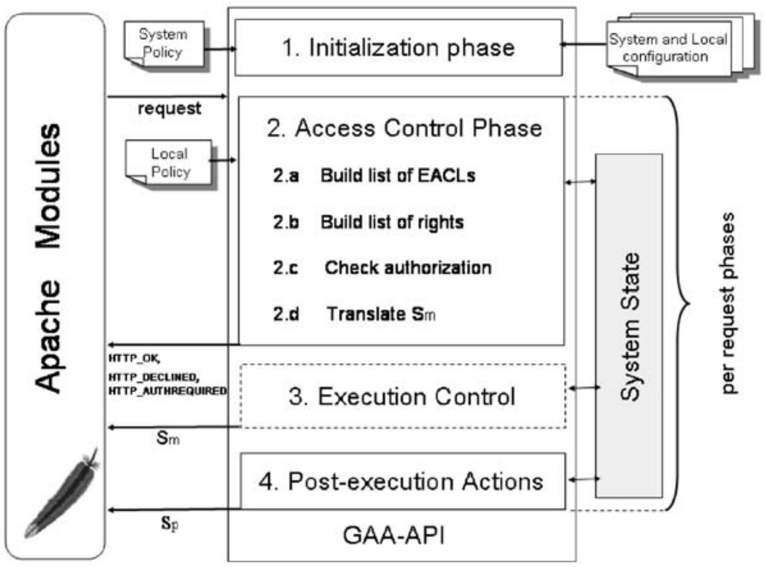

Fig. 2. GAA-Apache integration.

request is rejected; and `HTTP_AUTHREQUIRED`—user authentication is required to make further decision.

5.2 Adding GAA-API to Enhance the Access Control of the Apache Server

The preexisting version of Apache does not support flexible fine-grained policies that can control not only which users or groups and from which locations are allowed access, but also support other conditions, including time, system load, or system treat level. Within the Apache configuration file, the directive `Satisfy All` specifies that both of the constraints on IP address and user authentication should be satisfied to authorize an access request. `Satisfy Any` means that the request will be granted if either of the two constraints is met. However, these directives cannot express a policy with logical relations among three or more constraints. With our integration of the GAA-API, these limitations are eliminated. Here are the major advantages of the GAA-Apache integration:

- The GAA-API standard libraries provide routines that evaluate conditions on time, location, token-matching, etc. They can be used to check the access control parameters for Apache. For instance, server, client, and proxy IP address can be evaluated by the location routine. Client request time, creation time, and last modified time of requested resource can be evaluated by the time routine. Protocol version number and browser type can be evaluated by the token-matching routine.
- Besides making decisions of whether a request is accepted or rejected, the GAA-API libraries provide routines that can execute certain actions, such as logging information, notifying the administrator, etc. Furthermore, the routines can be activated whether the request succeeds or fails (when defined as request-result conditions) or whether the requested operation succeeds or fails (when defined as post-conditions). Thus, the GAA-API supports fine-tuning of the notification and audit services.
- The GAA-API is structured to support the addition of modules for evaluation of new conditions. Web masters can write their own routines to evaluate conditions or execute actions and register them with the GAA-API. Moreover, the routines can be loaded dynamically so that one does not need to recompile the whole Apache package to add new routines.
- The semantics of EACL format supported by the GAA-API can represent all logical combinations of security constraints.
- The GAA-API supports adaptive security policies, which detect security breaches and respond to attacks by modifying security measures automatically.

5.3 GAA-Apache Access Control

The GAA-API is integrated into Apache by modifying the *check_dir_access* function. The "glue" code extracts the information about requests from the Apache core modules, initializes the GAA-API, calls the API functions to evaluate policies and, finally, returns access control decision and status values to the modules. The GAA-Apache integration is shown in Fig. 2.

The GAA-API makes use of system-wide and local configuration and policy files. The configuration files list routines and parameters for evaluating conditions specified in the policy files. The system-wide policy applies to all applications in the system. The local policy file describes security requirements of Apache. The GAA-API returns three status values (`GAA_YES/GAA_NO/GAA_MAYBE`) to describe policy enforcement process:

1. Authorization status Sa indicates whether the request is authorized (`GAA_YES`), not authorized (`GAA_NO`), or uncertain (`GAA_MAYBE`).
2. Midcondition enforcement status Sm indicates the evaluation status of the midconditions.
3. Postcondition enforcement status Sp indicates the evaluation status of the postconditions.

The status values are obtained during the evaluation of conditions in the relevant EACL entries: GAA_YES—all conditions are met; GAA_NO—at least one of the conditions fails; GAA_MAYBE—none of the conditions fail, but there is at least one condition that is left unevaluated because the corresponding condition evaluation function is not registered with the API. Here are the three policy evaluation phases:

1. **Initialization phase**. When the server daemon of Apache starts, the GAA-API is initialized by calling *gaa_initialze* and *gaa_new_sc* to extract and register condition evaluation and policy retrieval routines from the system and local configuration files, fetch the system policy file, and generate internal structures for later use.
2. The **access control phase** starts with receiving a request to access an object (e.g., HTML file).

   a. The *gaa_get_object_policy_info* function is called to obtain the security policies associated with the requested object. The function reads the system-wide policy file, converts it to the internal EACL representation, and places it at the beginning of the list of EACLs. Next, the function retrieves and translates the local policy file and adds it to the list. The system and local policies are composed as described in Section 2.
   b. The request is converted into a list of requested rights. The context information (e.g., system configuration, server status, client status, and the details of access request) that may be used by the condition evaluation routines is extracted from the *request_rec* structure and is added to requested right structure as a list of parameters.
   c. Next, the *gaa_check_authorization* function is called to check whether the requested right is authorized by the ordered list of EACLs. This function finds the EACL entries where the requested right appears and calls the registered routines to evaluate pre and request-result conditions in the entries. If there are no preconditions, the authorization status is set to GAA_YES. Otherwise, the preconditions are evaluated and the result is stored in the authorization status Sa. If the request-result conditions are present in the entry, the conditions are evaluated and the intermediate result is calculated. The conjunction of the intermediate result and Sa is stored in the authorization status Sa.
   d. Finally, the status Sa is translated to the Apache format and is passed to the Apache core modules as a return value of the *check_dir_access* function. GAA_YES is translated to HTTP_OK (Apache can grant the request). GAA_NO is translated to HTTP_DECLINED (Apache should reject the request). In some cases, the GAA_MAYBE is translated to HTTP_AUTHREQUIRED, in other cases, to HTTP_DECLINED.

In particular, the GAA_MAYBE is used to enforce adaptive redirection policies. Apache may use the redirection for minimizing the network delay, load balancing, or security reason (e.g., redirect to a replica server that is closest to the client in terms of network distance). The redirection policies encoded in the preconditions specify, characteristics of a client, current system state, and URL that must serve the client. With this setup, the GAA-API first checks the preconditions that encode client's information and system state. The condition of type *pre_cond_redirect* encodes the URL and is returned unevaluated. When Apache receives the HTTP_AUTHREQUIRED, the server checks whether there is only one unevaluated condition of the type *pre_cond_redirect* and creates a redirected request using the URL from the condition value.

3. The **execution control phase** consists of starting the operation execution process and calling the *gaa_execution_control* function, which checks if the midconditions associated with the granted access right are met. The result is returned in Sm. The implementation of this phase has not been completed yet.
4. During the **postexecution action phase**, the *gaa_post_execution_actions* function is called to enforce the postconditions associated with the granted rights. This function performs policy enforcement after the operation completes by executing actions such as notifying by email, modifying system variables, writing log file, etc. The operation execution status (indicating whether the operation succeeded/failed) is passed to the *gaa_post_execution_actions*. If no postconditions are found, GAA_YES is returned; otherwise, the postconditions are evaluated and the result is returned in Sp.

6 DEPLOYMENTS

In this section, we describe several examples to illustrate how our framework can be deployed to enable fine-grained access control and intrusion detection and response.

6.1 Network Lockdown

We first show how our system adapts the applied authentication policies to require more information from a user when system threat level changes. Consider an organization with the mixed access to Web services. Access to some Web resources require user authentication, some do not. An IDS supplies a system threat level. For example, low threat level means normal system operational state, medium threat level indicates suspicious behavior, and high threat level means that the system is under attack. Policy: *When system threat level is higher than low, lock down the system and require user authentication for all accesses within the network.*
System-wide policy:

```
eacl_mode 1 # composition mode narrow
EACL entry 1
neg_access_right              *      *
pre_cond_system_threat_level  local  =high
```

Local policy:

```
# EACL entry 1
pos_access_right              apache   *
pre_cond_system_threat_level  local   > low
pre_cond_accessID_USER        apache   *
```

The system-wide policy specifies the mandatory requirement: "No access is allowed when system threat level is high" that cannot be bypassed by a local policy. The local policy specifies that all Apache accesses have to be authenticated if the system threat level is higher than "low." For example, if password authentication is required, a user will be asked for a username and a password.

6.2 Application-Level Intrusion Detection

We next show how the system supports prevention of penetration and/or surveillance attacks by detecting a CGI script abuse.

System-wide policy:

```
eacl_mode 1 # composition mode narrow
# EACL entry 1
neg_access_right        *     *
pre_cond_accessID_GROUP local BadGuys
```

Local policy:

```
# EACL entry 1
neg_access_right    apache *
pre_cond_regex      gnu    "'*phf*' '*test-cgi*'"
rr_cond_notify      local  on:failure/email:
                           sysadmin/info:CGIexploit
rr_cond_update_log  local  on:failure/BadGuys/info:IP

# EACL entry 2
pos_access_right apache *
```

Entry 1 in the system-wide policy specifies the mandatory requirement that members of the group BadGuys are denied access. Evaluation of the precondition `pre_cond_group` includes reading a log file of the suspicious IP addresses and trying to find an IP address that matches the address from which the request was sent. Entry 1 in the local policy contains a precondition `pre_cond_regex` that examines the request for occurrence of regular expressions `phf*` and `*test-cgi*`. If no match is found, the GAA-API proceeds to the next EACL entry that grants the request. If this condition is met, the request is rejected. The `rr_cond_notify` condition sends e-mail to the system administrator reporting time, IP address, URL attempted, and a threat type. Next, the `rr_cond_update_log` updates the group BadGuys to include new suspicious IP address from the request.

New signatures can be specified using regular expressions and numeric comparison. For example, the following precondition detects a particular DoS attack: `pre_cond_regex gnu` '*///////////////////*.' Evaluation of this condition includes checking the request for presence of a large number of "/" characters that most likely indicates an attempt to exploit a well-known apache bug that slows down Apache and fills up the logs fast.

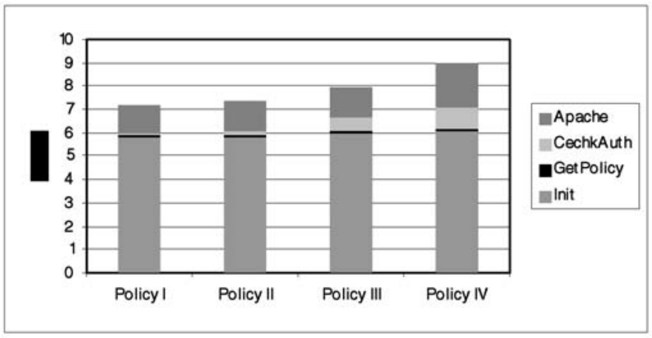

Fig. 3. Performance evaluation results.

The precondition `pre_cond_regex gnu` '*%*' detects malformed URLs (part of the URL contains the percent character). This may indicate ongoing attack, such as NIMDA. NIMDA exploits Microsoft IIS vulnerabilities by sending a malformed GET request. The precondition `pre_cond_expr local` $> 1{,}000$ checks that the length of input to a CGI script is no longer than 1,000 characters. This condition detects a buffer overflow attacks (e.g., Code Red IIS attack).

Adding suspicious hosts to the `BadGuys` may allow our system to stop attacks with unknown signatures. Often, vulnerabilities are tested by scripts that generate a number of requests. Each request exploits a particular bug. If the system identifies requests from an address as matching known attack signature, then subsequent requests from that host initiated by the same script, which checks for vulnerabilities not yet known, can still be blocked. Further, since this blacklist is specified in a system-wide policy, the list is shared by many of the hosts that improves the overall security of the system.

7 Performance Evaluation

The performance of GAA-API integrated Apache server was evaluated by using four different types of policy files. Policy I does not have any conditions and always grants access. Policy II includes simpleconditions that do not need any file access. Policy III includes conditions that require reading and writing variable files and log files. Policy IV contains more expensive conditions that check user authentication and perform asynchronous e-mail notification to the system administrator. The sample policy files can be found in the Appendix.

GAA-API function calls consist of three major phases: 1) "Initialization" phase that reads the configuration and system policy files for GAA-API, 2) "GetPolicy" phase that reads the local policy file associated with the object for which the access request is submitted, and 3) "CheckAuthorization" phase that returns authorization decision.

This experiment was conducted on a PC with an Intel Pentium 4, 1.8GHz, running RedHat Linux 7.3. Fig. 3 shows the result of the experiment. The values on the table are average values of 10 runs. The entry "Apache" is the execution time the original Apache modules incurred. The "Overhead" percentage was calculated based on the values in "Apache."

TABLE 1
Performance Evaluation Results

	Policy I	Policy II	Policy III	Policy IV
Init Phase	5.8432 ms	5.8469 ms	5.9445 ms	6.0472 ms
GetPolicy Phase	0.0805 ms	0.0919 ms	0.0957 ms	0.1051 ms
CheckAuth Phase	0.0241 ms	0.1332 ms	0.6401 ms	0.9731 ms
Apache	1.2348 ms	1.2779 ms	1.2960 ms	1.8570 ms *
Overhead with Init	481.68%	475.15%	515.46%	383.70%
Overhead w/o Init	8.47%	17.61%	56.77%	58.06%

* In Policy IV, GAA-API forks a new process that sends email asynchronously. Thus, the Apache process took around 50% more time (1.86 ms vs. 1.25 ms on the average) to run because of the child process for e-mail running in parallel.

As shown in the figure, "Initialization" is the most expensive phase. However, for each GAA-Apache process, initialization needs to be executed only once at the first time GAA-API is called. The figure shows the overheads that GAA-API introduces with the first request (Overhead with Init), and the overheads for the subsequent requests in each process (Overhead w/o Init).

The "Get Policy" phase is almost constant with low values because it just reads the local policy files. The only phase whose performance is affected by having different types of policies is the "Check Authorization" phase.

From Table 1, for the first call of GAA-API in a GAA-Apache process, GAA-API incurs an overhead of more than 400 percent because of the initialization phase. However, for the subsequentcalls of GAA-API in the same process, GAA-API skips the initialization phase and significantly reduces its overhead. For the policies with conditions that do not require file access (e.g., Policy II), the overhead from GAA-API function calls to the Apache Web server is lower than 20 percent. For more expensive policies with conditions that require file access, encryption, or process forking (e.g., Policies III and IV), the GAA-API's overhead was more than 50 percent.

The sample policy files used for the evaluation are fairly short in length, but we believe that they represent most of the possible cases. The individual policy files cannot grow huge because a local policy file is associated with an object or a group of objects. This means that, even if the system-wide policy could become complex, the performance of the system will not degrade linearly because the system will evaluate only the policy file that is specifically associated with the object for which the access request is submitted.

8 Related Work

AppShield [7] is a proprietary policy-based system that protects Web servers. The AppShield intercepts and analyzes all requests and dynamically adjusts its security policy to prevent attackers from exploiting application-level vulnerabilities. It uses dynamic policy not by looking for the signatures of suspicious behavior, but by knowing the intended behavior of the site and rejecting all other uses of the system. Emerald architecture [2] includes a data-collection module integrated with Apache Web server. The module extracts the request information internal to the Apache server and forwards it to an intrusion detection component that analyzes HTTP traffic. Both AppShield and Emerald systems are designed specifically for Web servers and cannot be used for other types of applications. In contrast, the GAA-API provides a generic policy evaluation and an application-level intrusion detection environment that can be used by different applications.

Almgren et. al. [1] provide an overview of the occurrences of Web server attacks and describe an intrusion detection tool that analyzes the CLF logs. The tool finds and reports intrusions by looking for attack signatures in the log entries. However, the monitor cannot directly interact with a Web server and, thus, cannot stop the ongoing attacks.

9 Discussion

Our application-level, policy-based approach to intrusion detection and response offers several important advantages over traditional host and network-based approaches:

1. *Customization.* Instead of having an IDS look for a restricted set of predefined signatures or time-variant statistical profiles, this approach allows each organization to define suspicious events in terms of policies for accessing application-level objects. The policies take into account the organization's and application's security requirements.
2. *Flexibility*. Security policies supported in our system can be defined in terms of acceptable and unacceptable access patterns to protected resources. For example: A *Closed World* policy states that everything that is not explicitly authorized is unacceptable and may indicate suspicious behavior. It might be possible to define a minimum set of ssh commands that are allowed and then define the presence of all other commands as a violation of the policy. An *Open World* policy defines that everything that is explicitly denied is unacceptable and may indicate suspicious behavior. A *Mixed World* policy may recognize some explicitly authorized access patterns as suspicious.
3. *Preemptive response.* By being integrated with the application and having the ability to control the three processing steps of a requested operation, the system can respond to suspected intrusion in real-time. For example, the system can deny the operation, suspend the operation execution, and notify about the success or failure of the completed operation.

4. *Elimination of several IDS vulnerabilities*. Traditional IDS is susceptible to desynchronization attacks since usually the IDS does not actively participate in the connection it monitors. With the proposed approach, the attempts to desynchronize the Detection engine from application will fail because the application is able to pass information to the engine through the GAA-API. As the system monitors events at the user level of abstraction, it is not vulnerable to traffic tampering attacks such as insertion and evasion. Fast attacks on IDS (that seek to exploit application's vulnerability before the IDS can apply counter measures) will not succeed because the system processes access requests by applications, and the application waits for the result.
5. *Reduction of false negatives and false positives*. The advantages of looking for the attacks at the application level include the ability to access decrypted information about a request. A request transported to the application through an encrypted channel is not visible to a network based IDS. The ability to interface with the application directly, with significant application-specific knowledge, allows application-based intrusion monitoring to detect suspicious behavior due to authorized users exceeding their authorization or exploitation of application-specific vulnerabilities. Using this approach could potentially result in detecting a custom attack that has never been observed in the past, thus reducing the number of false negatives. Another advantage is that information on how the request is handled by the server is available at the application level (e.g., whether the requested file is interpreted as a CGI script or HTML file). Both network and host-based IDSs could not make this distinction and if configured to look for strings matching "phf.cgi" and "test-cgi," they may produce false positives.

10 Future Work

To improve efficiency of the GAA-Apache integration, we will add support for caching of the retrieved and translated policies for later reuse by subsequent requests. We will investigate a possibility of implementing a simple profile building module and anomaly detector to support anomaly-based intrusion detection in addition to the signature-based. We plan to implement the execution control phase for Apache. We will explore the utility of midconditions for protection from compromised or badly written CGI scripts processed at the server. We plan to design a policy-controlled interface for establishing a subscription-based communication channels to extend the GAA-API and IDSs communication.

In this paper, we have considered simple attacks that require a single action (malicious request) in order to achieve the attacker's goal. More complex and stealthy attacks require a series of actions that constitute an attack scenario. In order to detect such attacks, we will extend our system with the support for attack signatures that describe a sequence of access requests and system state conditions that represent an attack. To implement detection of such complex signatures, we will use hypothesis generation techniques. In particular, we will study the application of Bayesian methods [4] to classify observed events into attack scenarios.

In the current framework, we assume that conditions are evaluated consecutively and that authorization requests do not overlap. These two assumptions enable us to concentrate on a single condition evaluation per each time interval and, therefore, avoid the problem of coordination of multiple condition evaluation processes. However, this approach results in inefficient policy evaluation process and leads to systems that cannot scale to large numbers of objects. The future directions for this research include exploring extensions to the framework to support: concurrent requests, replication of the evaluation mechanism, concurrent evaluation of conditions within the same request, and distributed policy enforcement. At this point, the issues of spatial and temporal relationships among the policy computations become critical. Policies that govern the same object may have nontrivial interdependencies which must be carefully analyzed and understood.

Another limitation of the current framework is reliance on a policy administrator for defining condition evaluation order, which is then enforced by the framework. The limited awareness of the spatial and temporal dependencies among security policies may cause inconsistencies and undesirable system behavior. In many cases, administrators may not have a clear picture of the ramifications of policy enforcement actions; therefore, enforcing these policies might have unexpected interactive or concurrent behavior. Automation is essential to minimize human error, and it can only be used safely when there is a formal model that explicitly addresses both the spatial and the temporal aspects of dynamic authorization. Much research has been done in the area of integration of active mechanisms into relational and object-oriented DBMSs. We plan to test the applicability of methods and concepts from the field of active database systems to develop static and dynamic analysis techniques for adaptive policies. The reuse of techniques developed in the database community is necessary to apply best practices and to avoid repeating mistakes.

Finally, in order to put the developed formalism into practice, the researchers will implement a set of tools that provide graphical interfaces supporting both static activities such as:

- **A specialized interactive policy analyzer/editor**—a development tool that provides compile-time examining and detection of policy rule problems. The tool will be used to create policies with strong security guarantees, eliminating guesswork in the design, and deployment of dynamic authorization.
- **A runtime monitor** that provides runtime support for the execution rules derived from the semantic restrictions to maintain the policy processing automatically, asynchronously, and correctly.

11 Conclusions

Traditional access control mechanisms have little ability to support or respond to the detection of attacks. In this paper, we presented a generic authorization framework that supports security policies that can detect attempted and actual security breaches and which can actively respond by modifying security policies dynamically. The GAA-API combines policy enforcement with application-level intrusion detection and response, allowing countermeasures to be applied to ongoing attacks before they cause damage. Because the API processes access control request by applications, it is ideally placed to apply application-level knowledge about policies and activities to identify suspicious activity and apply appropriate responses. The GAA-API implementation is available at http://gaaapi.sysproject.info. The API has been integrated with several applications, including Apache, SOCKS5, sshd, and FreeS/WAN IPsec for Linux.

Appendix

The four different types of policies used in the Section 7.
Policy I

```
pos_access_right apache *
```

Policy II

```
pos_access_right     apache *
pre_cond_access_host apache "127.0.0.1 OR 128.9.0.0/16
                            OR usc.edu"
pre_cond_access_time apache "01/01/03-12/31/05
                            MON-FRI"
pre_cond_check_regex apache "#apache.uri =' *.html' "
```

Policy III

```
neg_access_right     apache  *
pre_cond_check_equal apache  "%(#remote_ip.threat
                                level) = HIGH"
rr_cond_inc_variable apache  "%(#remote_ip.reject
                                count)"
rr_cond_append_log   apache  "%LogMsgReject"
```

Policy IV

```
pos_access_right            apache  *
pre_cond_access_user        apache  "%InspectedUser
                                     List"
rr_cond_async_email_notify  apache  "root@localhost"
```

Acknowledgments

This effort was sponsored by the US Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, US Air Force Materiel Command, USAF, and the Xerox Corporation under the following agreements: 1) F30602-00-2-0595, Dynamic Policy Evaluation of Containing Network Attacks Project (DEFCN); 2) DABT63-94-C-0034, Security Infrastructure for Large Distributed Systems Project (SILDS); 3) J-FBI-95-204, Global Operating System Technologies Project (GOST); 4) DE-FC03-99ER25397, Diplomat project; and 5) HE1254-97, XAUTH Project. The US Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the US Defense Advanced Research Projects Agency (DARPA), the US Air Force Research Laboratory, US Department of Energy, the US Government, or the Xerox Corporation. Figures and descriptions are provided by the authors and are used with permission.

References

[1] M. Almgren, H. Debar, and M. Dacier, "A Lightweight Tool for Detecting Web Server Attacks," *Proc. Network and Distributed System Security Symp.*, 2000.
[2] M. Almgren and U. Lindqvist, "Application-Integrated Data Collection for Security Monitoring," *Proc. Fourth Int'l Symp. Recent Advances in Intrusion Detection,* pp. 22-36, 2001.
[3] R. Bace and P. Mell, "Intrusion Detection Systems," *NIST Special Publication on Intrusion Detection Systems,* Nat'l Inst. of Standards and Technology, 2001.
[4] D.J. Burroughs, L.F. Wilson, and G.V. Cybenko, "Analysis of Distributed Intrusion Detection Systems Using Bayesian Methods," *Proc. IEEE Int'l Performance Computing and Comm. Conf.*, Apr. 2002.
[5] T.V. Ryutov and B.C. Neuman, "The Specification and Enforcement of Advanced Security Policies," *Proc. Conf. Policies for Distributed Systems and Networks,* 2002.
[6] R. Thau, "Design Considerations for the Apache Server API," *Proc. Fifth Int'l World Wide Web Conf.*, 1996.
[7] Sanctum, Inc., http://www.sanctuminc.com, 2003.

Tatyana Ryutov received the MS degree in applied mathematics from Moscow State University, Russia, in 1991, and the MS and PhD degrees in computer science from the University of Southern California, USC, in 1999 and 2002, respectively. She joined USC/ISI in 1996 working as a graduate research assistant, and focused on the development and implementation of the access control framework for distributed systems that supports active policies, policy composition, and is sensitive to network threat conditions. Currently, Dr. Ryutov is working as a computer scientist at the University of Southern California's Information Sciences Institute with Dr. Clifford Neuman on the Dynamic Policy Evaluation for Containing Network Attacks (DEFCN) project.

Clifford Neuman received the bachelors degree from the Massachusetts Institute of Technology and, subsequently, worked at Project Athena. He received the MS and PhD degrees from the University of Washington. He is the director of the Center for Computer Systems Security at The Information Sciences Institute (ISI) of the University of Southern California (USC), associate division director of the Computer Networks Division at ISI, and a faculty member in the Computer Science Department at USC. Dr. Neuman conducts research in distributed systems, computer security, and electronic commerce. He is the principal designer of Kerberos authentication system, which, among other deployments, provides user authentication for Microsoft's Windows 2000 and Windows XP. He also developed the NetCheque® and NetCash systems, and the Prospero Directory Service. His current research focuses on the use of dynamic security policies in distributed systems that can support the formation of dynamic coalitions of cooperating organizations while adapting and responding to perceived network threats. He is a senior member of the IEEE.

Dongho Kim received the BS degree in computer engineering from Seoul National University in 1990, the MS degree in computer science from the University of Southern California (USC) in 1992, and the PhD degree in computer science from USC in 2002. He is a computer scientist at the University of Southern California's Information Sciences Institute (USC/ISI). He has been working on the Dynamic Policy Evaluation for Containing Network Attacks (DEFCN) project for three years as a member of Global Operating Systems Technology (GOST) group in Computer Networks Division of USC/ISI. He has been an instructor for the graduate-level Advanced Operating Systems course at the USC during Fall semesters since 2001. He is a member of the IEEE and the IEEE Computer Society.

Li Zhou received the BS degree in computer science from Beijing University in 2001. He is a PhD student in the Computer Science Department, University of Southern California (USC). Currently, he is working in the Generic Operating System Technology (GOST) Group, Information Science Institute, USC, as a graduate researching assistant.

Appendix B:

Integrated Access Control and Intrusion Detection for Web Servers[*]

Tatyana Ryutov, Clifford Neuman, Dongho Kim and Li Zhou
Information Sciences Institute
University of Southern California
{tryutov, bcn, dongho, zhou}@isi.edu

Abstract

Current intrusion detection systems work in isolation from access control for the application the systems aim to protect. The lack of coordination and inter-operation between these components prevents detecting and responding to ongoing attacks in real time, before they cause damage. To address this, we apply dynamic authorization techniques to support fine-grained access control and application level intrusion detection and response capabilities. This paper describes our experience with integration of the Generic Authorization and Access Control API (GAA-API) to provide dynamic intrusion detection and response for the Apache Web Server. The GAA-API is a generic interface which may be used to enable such dynamic authorization and intrusion response capabilities for many applications.

1 Introduction and Motivation

Web servers continue to be attractive targets for attackers seeking to steal or destroy data, deny user access or embarrass organizations by changing web site contents. Furthermore, because web servers must be publicly available around the clock, the server is an easy target for outside intruders. In order to penetrate their targets, attackers may exploit well-known service vulnerabilities. A web server can be subverted through vulnerable CGI scripts, which may be exploited by meta characters or buffer overflow attacks. These vulnerabilities may be related to the default installation of the server or may be introduced by careless writing of custom scripts.

Web servers are also popular targets for Denial of Service (DoS) attacks. An attacker sends a stream of connection requests to a server in an attempt to crash or slow down the service. Launching a DoS attack against a web server can be accomplished in many ways, including ill-formed HTTP requests (e.g., a large number of HTTP headers). As the server tries to process such requests it slows down and becomes unable to process other requests. In addition, web servers exhibit susceptibility to password guessing attacks.

To address these risks, web servers require increased security protection. Effective system security starts with security policies that are supported by an access control mechanism. Access control policy to be enforced should depend on the current state of the system, e.g., time of day, system load or system threat level. More restrictive organizational policies may be enforced after hours, when the system is busy or if suspicious activity has been detected. Unfortunately, many web servers (e.g., Apache and IIS) support only limited identity- and host-based policies that deny/allow access to protected resources. The policies are checked only when an access request is received to determine whether the request should be permitted or forbidden. These policies do not support observing and reporting suspicious activity (e.g., embedding hexadecimal characters in a query) and modifying system protection as a result.

Thus, the security policies must not only specify legitimate user privileges but also aid in the detection of threats and adapt their behavior based on perceived system threat conditions. Even a single instance of a request for a vulnerable CGI script or malformed request should be reported immediately and countermeasures should be applied. Such countermeasures may include:
- generating audit records;
- notifying network servers that are monitoring security rel-

[*]Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, and the Xerox Corporation under the following agreements: (1) F30602-00-2-0595, Dynamic Policy Evaluation of Containing Network Attacks Project (DEFCN); (2) DABT63-94-C-0034, Security Infrastructure for Large Distributed Systems Project (SILDS); (3) J-FBI-95-204, Global Operating System Technologies Project (GOST); and (4) DE-FC03-99ER25397, Diplomat project. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, U.S. Department of Energy or the U.S. Government. Figures and descriptions were provided by the authors and are used with permission.

evant events in the system;
- tightening local policies (e.g., restricting access to local users only or requesting extra credentials);
- modifying overall system protection. Examples include terminating the session, logging the user off the system, disabling local account or blocking connections from particular parts of the network or stopping selected services (e.g., disable ssh connections).

These actions would be followed by an alert to the security administrator, who can then assess the situation and take the appropriate corrective actions. This step is important, since an automated response to attacks can be used by an intruder in order to stage a DoS (the intruder could have impersonated a host or a user).

Traditional access control mechanisms were not designed to aid the detection of threats or to adjust their behavior based on perceived threat conditions. Common countermeasures to web server threats depend on separate components like firewalls, Intrusion Detection Systems (IDSs), and code integrity checkers. While these components are useful in detecting some kinds of attacks, they do not fully address a web server's security needs. For example, firewalls can deny access to unauthorized network connections, but they can not stop attacks coming in via authorized ports. In the general case, IDSs provide only incomplete coverage, leaving sophisticated attacks undetected. Other disadvantages include: large number of false positives and inability to preemptively respond to attacks. Integrity checkers can detect unauthorized changes to files on a web site, but only after the damage has been done.

Motivated by the multitude of web server vulnerabilities and generally unsatisfactory server protection, we propose integrated approach to web server security - the Generic Authorization and Access-control API (GAA-API) that supports fine-grained access control and application level intrusion detection and response.

The GAA-API evaluates HTTP requests and determines whether the requests are allowed and if they represent a threat according to a policy. Our approach differs from other work done in this area by supporting access control policies extended with the capability to identify (and possibly classify) intrusions and respond to the intrusions in real time. The policy enforcement takes three phases:
1. Before requested operation (e.g., display an HTML file or run a CGI program) starts; to decide whether this operation is authorized.
2. During the execution of the authorized operation; to detect malicious behavior in real-time (e.g., a user process consumes excessive system resources).
3. When the operation is completed; to activate post execution actions, such as logging and notification whether the operation succeeds/fails. For example, alerting that a particular critical file (e.g., /etc/passwd) was modified can trigger a process to check the contents of the file (e.g., check for a null password).

By being integrated with the web server and having the ability to control the three processing steps of the requested operation, the GAA-API can respond to suspected intrusion in real-time before it causes damage, whether it is site defacement, data theft or a DoS attack.

The disadvantage of the proposed approach is that a web server has to be modified in order to utilize the GAA-API. However, once the relatively easy integration is completed, it becomes possible to handle access control decisions and application level intrusion detection simultaneously. Furthermore, since the GAA-API is a generic tool, it can be used by a number of different applications with no modifications to the API code. In this paper we focus on the web server. However, the API can provide enhanced security for applications with different security requirements. We have integrated the GAA-API with Apache web server, sshd and FreeS/WAN IPsec for Linux.

2 Policy Representation

The Extended Access Control List (EACL) is a simple language that we implemented to describe security policies that govern access to protected resources, identify threats that may occur within application and specify intrusion response actions. An EACL is associated with an object to be protected and specifies positive and negative access rights with optional set of associated conditions that describe the context in which each access right is granted or denied. An EACL describes more than one set of disjoint policies. The policy evaluation mechanism is extended with the ability to read and write system state. The implementation is based on conditions that provide support for monitoring and updating internal system structures and their runtime behaviors.

A condition may either explicitly list the value of a constraint or specify where the value can be obtained at run time. The latter allows for adaptive constraint specification, since alowble times, locations and thresholds can change in the event of possible security attacks. The value of condition can be supplied by other services, e.g., an IDS.

In our framework, all conditions are classified as:

1. **pre-conditions** specify what must be true in order to grant or deny the request, e.g., `access identity`, `time`, `location` and `system threat level`.

2. **request-result** conditions must be activated whether the authorization request is granted or whether the request is denied, e.g., `audit` and `notification`.

3. **mid-conditions** specify what must be true during the execution of the requested operation, e.g., a CPU usage `threshold` that must hold during the operation execution.

4. **post-conditions** are used to activate post execution actions, such as logging and notification whether the operation succeeds/fails.

Failure of some of these conditions may signal suspicious behavior,e.g., access is requested at unexpected times or unusual locations. Some conditions can trigger defensive measures in response to a perceived system threat level, e.g., impose a limit on resource consumption or increase auditing.

A **condition block** defines a conjunction of a totally ordered set of conditions. Conditions are evaluated in the order they appear within a condition block.

An **EACL entry** consists of a positive or negative access right and four optional condition blocks: a set of preconditions, a set of request-result conditions, a set of midconditions and a set of post-conditions.

An **EACL** consists of an ordered set of disjunctive EACL entries. An EACL representation supports disjunction and conjunction of conditions to activate different control modes. A transition between the disjoint EACL entries is regulated automatically by reading the system state (e.g., time of day or the system threat level). Detailed EACL syntax is given in the Appendix.

In the current framework, the evaluation of entries within an EACL and evaluation of conditions within an EACL entry is totally ordered. Evaluation of an EACL starts from the first to the last in the list of EACL entries. The resolution of inconsistent authorizations is based on ordering. The entries which already have been examined take precedence over new entries.

The order has to be assessed before EACL evaluation starts. Determining the evaluation order is currently done by a policy officer. We recognize that the function of defining the order of EACL entries and conditions within an entry can be best served by an automated tool to ensure policy correctness and consistency and to ease the policy specification burden on the policy officer. We plan to design and implement such tool in the future. For further details about the authorization model see [4].

The GAA-API provides a general-purpose execution environment in which EACLs are evaluated.

2.1 Policy Composition

Policy Composition is a process of relating separately specified policies. Our framework supports system-wide and local policies. This separation is useful for efficient policy management. Instead of repeating policies that apply to all applications in individual application policies, we define these policies as a separate *system-wide policy* that is applied globally and is consulted on all the accesses to all applications. *Local policies* allow users and applications to define their own policy in addition to the global one.

The composed policy is constructed by merging the system-wide and local policies. First, system-wide policies are retrieved and placed at the beginning of the list of policies. Then the local policies are retrieved and added to the list. Thus, system-wide policies implicitly have higher priority than the local policies.

A system-wide policy specifies a **composition mode** that describes how local policies are to be composed with the system-wide policy. The framework supports three composition modes:

expand

A system-wide policy broadens the access rights beyond those granted by local policies. It is the equivalent of a disjunction of the rights. The access is allowed if either the system-wide or the local policy allows the access. This is useful to ensure that a request permitted by the system-wide policy can not fail due to access rejection at the local level.

narrow

A system-wide policy narrows the access rights so that objects can not be accessed under particular conditions regardless of the local policies. The policy that controls access to an object may have mandatory and discretionary components. Generally, mandatory policy is set by the domain administrator, while discretionary policy is set by individuals or applications. The mandatory policies must always hold. The discretionary policies must be satisfied in addition to the mandatory policies. Thus, the resulting policy represents the conjunction of the mandatory and discretionary policies.

stop

If a system-wide policy exists, that policy is applied and local policies are ignored. An administrator may require complete overriding of the local policies with the system-wide policies. This is useful in order to react quickly to an attack. One might use the *stop* mode to shut down certain component systems. This is also useful when the administrator wants to, for example, allow access to a document (e.g., a system log file) only to himself. If he specifies a policy using the *expand* mode, then additional access can be granted at the local level. If he uses *narrow* mode, the local policies could add additional restrictions that can deny the access.

To evaluate several separately specified local (or system-wide) policies, we take a conjunction of the policies.

3 GAA-API and IDS interactions

The data extracted from an application at the access control time can be supplemented with data from a network- and host-based IDSs to detect attacks not visible at the application level and reduce false alarm rate.

The current GAA-API interaction with an IDS is limited to determining the current system threat profile and adapting the security policy to respond to changing security requirements. Our next task is to support closer interaction between the GAA-API and different IDSs. Here are the kinds of information[1] that the GAA-API can report to IDS:

1. Ill-formed access requests, which may signal an attack. Because the GAA-API processes access requests by applications, the API can apply application level knowledge to determine whether the request is properly formed.

2. Accesses requests with parameters that are abnormally large or violate site's policy.

3. Access denial to sensitive system objects.

4. Violating threshold conditions, e.g., the number of failed login attempts within a given period of time.

5. Detected application level attacks. The report may include threat characteristics, such as attack type and severity, confidence value and defensive recommendations.

6. Unusual or suspicious application behavior such as creating files.

7. Legitimate access request patterns. This information can be used to derive profiles that describe typical behavior of users working with different applications.

The GAA-API can request a network-based IDS to report, for example, indications of address spoofing. This information can be used in addition to the application level attack signatures to further reduce the false positive rate and avoid DoS attacks. This is particularly important for applying pro active countermeasures, such as updating firewall rules and dropping connections.

The API can request information for adjusting policies, such as values for thresholds, times and locations. The values may depend on many factors and can be determined by a host-based IDS and communicated to the GAA-API.

4 The Apache Access Control

Apache's access control system provides a method for web masters to allow or deny access to certain URL paths, files, or directories. Access can be controlled by requiring username and password information or by restricting the originating IP address of the client request. Access control is usually confined to specific directories of the document tree. When processing client's request to access a document Apache looks for an access control file called .htaccess in every directory of the path to the document. Here is a sample .htaccess file:

```
Order Deny, Allow
Deny from All
Allow from 10.0.0.0/255.0.0.0
AuthType Basic
AuthUserFile /usr/local/apache2/.htpasswd-isi-staff
Require valid-user
Satisfy All
```

The "Allow from 10.0.0.0/255.0.0.0" allows connections only from hosts within the specified IP range. All other hosts will get a "Permission Denied" message. The "Require valid-user" requires that the user enter a username and password. These username/password pairs are stored in a separate file specified by the "AuthUserFile" directive.

5 Adding GAA-API to Enhance the Access Control of the Apache Server

Unfortunately, the current version of Apache does not support flexible fine-grained policies. Within the Apache configuration file, the directive *Satisfy All* specifies that both of the constraints on IP address and user authentication should be satisfied to authorize an access request. *Satisfy Any* means that the request will be granted if either of the two constraints is met. However, these directives can not express a policy with logical relations among three or more constraints. Therefore, new semantics must be introduced to specify a more flexible access control policy. Here are the major advantages of the integration:

1. Besides making decisions of whether a request is accepted or rejected, the GAA-API libraries provide routines that can execute certain actions, such as logging information, notifying administrator, etc. Furthermore, the routines can be activated whether the request succeeds/fails (when defined as request-result conditions) or whether the requested operation succeeds/fails (when defined as post-conditions). Thus, the GAA-API supports fine-tuning of the notification and audit services.

2. The GAA-API is structured to support the addition of modules for evaluation of new conditions. Web masters can write their own routines to evaluate conditions or execute actions and register them with the GAA-API. Moreover, the routines can be loaded dynamically so that one does not need to recompile the whole Apache package to add new routines.

3. The semantics of EACL format supported by the GAA-API can represent all logical combinations of security constraints.

4. The GAA-API supports adaptive security policies, which detect security breaches and respond to attacks by modifying security measures automatically.

[1] This information can be used locally by modules that implement the application level intrusion/misuse detection, as described in Section 7 and/or forwarded the information to IDSs for analysis.

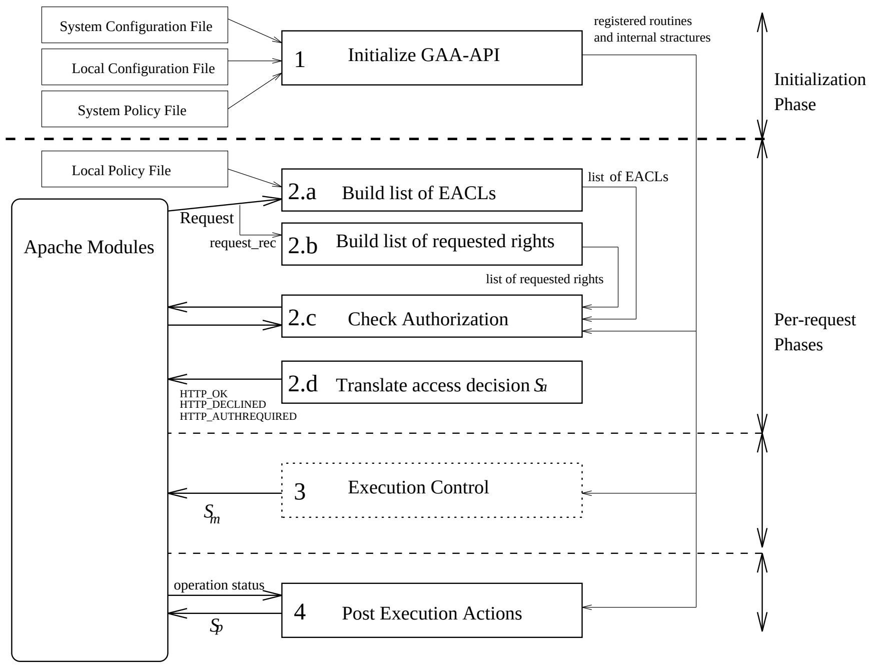

Figure 1. GAA-Apache integration

6 GAA-Apache Access Control

The GAA-API is integrated into Apache by modifying the $check\_dir\_access$ function. The "glue" code extracts the information about requests from the Apache core modules, initializes the GAA-API, calls the API functions to evaluate policies, and finally returns access control decision and status values to the modules. The GAA-Apache integration is shown in Figure 1. The GAA-API makes use of system-wide and local configuration and policy files. The configuration files list routines and parameters for evaluating conditions specified in the policy files. The system-wide policy applies to all applications in the system. The local 3 status values:
1. authorization status S_a indicates whether the request is authorized, not authorized or uncertain.
2. mid-condition enforcement status S_m.
3. post-condition enforcement status S_p.

The status values ($GAA\_YES/GAA\_NO/GAA\_MAYBE$) are obtained during the evaluation of conditions in the relevant EACL entries:
GAA_YES - all conditions are met;
GAA_NO - at least one of the conditions fails;
GAA_MAYBE - none of the conditions fails but there is at least one condition that is left unevaluated. The GAA-API returns GAA_MAYBE if the corresponding condition evaluation function is not registered with the API.

1. **Initialization phase**. When the server daemon of Apache starts, first the GAA-API is initialized by calling $gaa\_initialze$ and gaa_new_sc that extract and register condition evaluation and policy retrieval routines from the system and local configuration files, fetch the system policy file, and generate internal structures for later use.

2. The **access control phase** starts with receiving a request to access an object (e.g., HTML file or CGI script).

   (a) The $gaa\_get\_object\_policy\_info$ function is called to obtain the security policies associated with the requested object. The function reads the system-wide policy file, converts it to the internal EACL representation and places it at the beginning of the list of EACLs. Next, the function retrieves and translates the local policy file and adds it to the list. The system and local policies are composed as described in Section 2.1.

   (b) The request is converted into a list of requested rights. The context information (e.g., system configuration, server status, client status and the details of access request) that may be used by the condition evaluation routines is extracted from the $request\_rec$ structure and is added to requested right structure as a list of parameters. These parameters are classified with "type" and "authority" so that GAA-API routines that evaluate conditions with the same type and authority could find the relevant parameters.

(c) Next, the $gaa\_check\_authorization$ function is called to check whether the requested right is authorized by the the ordered list of EACLs. This function finds the EACL entries where the the requested right appears and calls the registered routines to evaluate pre- and request-result conditions in the entries. If there are no pre-conditions, the authorization status is set to GAA_YES. Otherwise, the pre-conditions are evaluated and the result is stored in the authorization status S_a.

If the request-result conditions are present in the entry, the conditions are evaluated and the intermediate result is calculated. The conjunction of the intermediate result and S_a is stored in the authorization status S_a.

(d) Finally, the status S_a is translated to the Apache format and is passed to the Apache core modules as a return value of the $check\_dir\_access$ function. GAA_YES is translated to $HTTP\_OK$ (Apache can grant the request). GAA_NO is translated to $HTTP\_DECLINED$ (Apache should reject the request). In some cases, the GAA_MAYBE is translated to $HTTP\_AUTHREQUIRED$, in other cases to $HTTP\_DECLINED$.

In particular, the GAA_MAYBE is used to enforce adaptive redirection policies. Apache may use the redirection for minimizing the network delay, load balancing or security reasons. For example, redirect to a replica server that is closest to the client in terms of network distance. The redirection policies encoded in the pre-conditions specify, characteristics of a client, current system state and URL that must serve the client. With this setup, the GAA-API first checks the pre-conditions that encode client's information and system state. The condition of type `pre_cond_redirect` encodes the URL and is returned unevaluated. When Apache receives the $HTTP\_AUTHREQUIRED$, the server checks whether there is only one unevaluated condition of the type `pre_cond_redirect` and creates a redirected request using the URL from the condition value.

3. The **execution control phase** consists of starting the operation execution process and calling the $gaa\_execution\_control$ function which checks if the mid-conditions associated with the granted access right are met. The result is returned in S_m. The implementation of this phase has not been completed yet.

4. During the **post-execution action phase** the $gaa\_post\_execution\_actions$ function is called to enforce the post-conditions associated with the granted rights. This function performs policy enforcement after the operation completes by executing actions such as notifying by email, modifying system variables, writing log file, etc. The operation execution status (indicating whether the operation succeeded/failed) is passed to the $gaa\_post\_execution\_actions$. If no post-conditions are found, GAA_YES is returned, otherwise the post-conditions are evaluated and the result is returned in S_p.

7 Deployments

In this section we describe several examples to illustrate how our framework can be deployed to enable fine-grained access control and intrusion detection and response.

7.1 Network Lockdown

We first show how our system adapts the applied authentication policies to require more information from a user when system threat level changes. Consider an organization with the following characteristics:

- Mixed access to web services. Access to some web resources require user authentication, some do not.

- An IDS supplies a system threat level. For example, low threat level means normal system operational state, medium threat level indicates suspicious behavior and high threat level means that the system is under attack.

- Policy: *When system threat level is higher than low, lock down the system and require user authentication for all accesses within the network.* Strong authentication protects against outside intruders. To some extent, authentication may help to reduce insider misuse. In particular, insiders are discouraged if the identity of a user can be established reliably.

System-wide policy:
```
eacl_mode 1 # composition mode narrow
# EACL entry 1
neg_access_right * *
pre_cond_system_threat_level local =high
```

Local policy:
```
# EACL entry 1
pos_access_right apache *
pre_cond_system_threat_level local >low
pre_cond_accessID_USER apache *
```
The system-wide policy specifies mandatory requirement "No access is allowed when system threat level is high" that

can not be bypassed by a local policy. The local policy specifies that all Apache accesses have to be authenticated if the system threat level is higher than “low”.

7.2 Application level Intrusion Detection

We next show how the system supports prevention of penetration and/or surveillance attacks by detecting a CGI script abuse.
System-wide policy:

```
eacl_mode 1 # composition mode narrow
# EACL entry 1
neg_access_right * *
pre_cond_accessID_GROUP local BadGuys
```

Local policy:

```
# EACL entry 1
neg_access_right apache *
pre_cond_regex gnu ‘‘’*phf*’ ’*test-cgi*’’’
rr_cond_notify local
on:failure/email:sysadmin/info:CGIexploit
rr_cond_update_log local
on:failure/BadGuys/info:IP
# EACL entry 2
pos_access_right apache *
```

Entry 1 in the system-wide policy specifies mandatory requirement that members of the group `BadGuys` are denied access. Evaluation of the pre-condition `pre_cond_group` includes reading a log file of the suspicious IP addresses and trying to find an IP address that matches the address the request was sent from. Entry 1 in the local policy contains a pre-condition `pre_cond_regex` that examines the request for occurrence of regular expressions `*phf*` and `*test-cgi*`. If no match is found, the GAA-API proceeds to the next EACL entry that grants the request.

If this condition is met, the request is rejected.
The `rr_cond_notify` condition sends e-mail to the system administrator reporting time, IP address, URL attempted and a threat type.
Next, the `rr_cond_update_log` updates the group `BadGuys` to include new suspicious IP address from the request.

New signatures can be specified using regular expressions and numeric comparison. For example, the following pre-condition detects a particular DoS attack:
`pre_cond_regex gnu ’*////////////////////*’`
Evaluation of this condition includes checking the request for presence of a large number of ”/” characters that most likely indicates an attempt to exploit a well-known apache bug that slows down Apache and fills up logs fast.

The pre-condition `pre_cond_regex gnu ’*%*’` detects malformed URLs (part of the URL contains the percent character). This may indicate ongoing attack, such as NIMDA. NIMDA exploits Microsoft IIS vulnerabilities by sending a malformed GET request.

The pre-condition `pre_cond_expr local >1000` checks that the length of input to a CGI script is no longer than 1000 characters. This condition detects a buffer overflow attacks, e.g., Code Red IIS attack.

Adding suspicious hosts to the `BadGuys` may allow our system to stop attacks with unknown signatures. Often vulnerabilities are tested by scripts that generate a number of requests. Each request exploits a particular bug. If the system identifies requests from an address as matching known attack signature, then subsequent requests from that host (initiated by the same script), checking for vulnerabilities we might not yet know about, can still be blocked. Further, since this blacklist is specified in a system-wide policy, the list is shared by many of our hosts that improves security of the system overall.

8 Performance

In our experiment, we used the system-wide and local policy files shown in Sections 7.1 and 7.2, respectively. The experiment was performed 20 times on a PC with an Intel 1.8GHz Pentium 4 CPU, running RedHat Linux v7.1.

On average, GAA-API functions took 5.9 milliseconds (ms) without email notification (53.3 ms with email notification) while running Apache functions including GAA-API functions took 19.4 ms (66.8 ms with email notification). The overhead introduced by the GAA-API is 30% if email notification is not taken into account. If the email notification is enabled, the overhead increases to 80%.

9 Implementation Status and Future Work

The GAA-API implementation is available at http://www.isi.edu/gost/info/gaaapi/source.
The API has been integrated with several applications, including Apache, sshd and FreeS/WAN IPsec for Linux.

To improve efficiency of the GAA-Apache integration we will add support for caching of the retrieved and translated policies for later reuse by subsequent requests. We will investigate a possibility of implementing a simple profile building module and anomaly detector (implemented using conditions) to support anomaly-based intrusion detection in addition to the signature-based.

We plan to implement the execution control phase for Apache. We will explore the utility of mid-conditions for protection from untrusted downloaded code, such as Java applets and Netscape plug-ins. The mid-conditions will control actions of the downloaded content on a client machine throughout the execution of the content.

We plan to design a policy-controlled interface for establishing a subscription-based communication channels to allow GAA-API and IDSs to communicate.

10 Related Work

AppShield [5] is a proprietary policy-based system that protects web servers. The AppShield intercepts and analyzes all requests and dynamically adjusts its security policy to prevent attackers from exploiting application-level vulnerabilities. It uses dynamic policy not by looking for the signatures of suspicious behavior but by knowing the intended behavior of the site and rejecting all other uses of the system.

Emerald architecture [2] includes a data-collection module integrated with Apache Web server. The module extracts the request information internal to the Apache server and forwards it to an intrusion detection component that analyzes HTTP traffic.

Both AppShield and Emerald systems are designed specifically for the web servers and can not be used for other types of applications. In contrast, the GAA-API provides a generic policy evaluation and an application-level intrusion detection environment that can be used by different applications.

Almgren, et. al., [1] provide an overview of the occurrences of web server attacks and describe an intrusion detection tool that analyzes the CLF logs. The tool finds and reports intrusions by looking for attack signatures in the log entries. However, the monitor can not directly interact with a web server and, thus, can not stop the ongoing attacks.

11 Conclusions

Traditional access control mechanisms have little ability to support or respond to the detection of attacks. In this paper we presented a generic authorization framework that supports security policies that can detect attempted and actual security breaches and which can actively respond by modifying security policies dynamically. The GAA-API combines policy enforcement with application-level intrusion detection and response, allowing countermeasures to be applied to ongoing attacks before they cause damage. Because the API processes access control request by applications, it is ideally placed to apply application-level knowledge about policies and activities to identify suspicious activity and apply appropriate responses.

12 Appendix

We use the Backus-Naur Form to denote the elements of our EACL language. Items inside round brackets, `()` are optional. Curly brackets, `{}`, surround items that can repeat zero or more times. A vertical line, `|`, separates alternatives. Items inside double quotes are the terminal symbols. An EACL is specified according to the following format:

```
eacl ::= (composition_mode) {entry}
entry ::= pright conds | nright
pre_cond_block rr_cond_block
pright ::= "pos_access_right" def_auth value
nright ::= "neg_access_right" def_auth value
conds ::= pre_cond_block rr_cond_block
mid_cond_block post_cond_block
pre_cond_block ::= {condition}
rr_cond_block ::= {condition}
mid_cond_block ::= {condition}
post_cond_block ::= {condition}
condition ::= cond_type def_auth value
composition_mode ::= "0"|"1"|"2"
cond_type ::= alphanumeric_string
def_auth ::= alphanumeric_string
value ::= alphanumeric_string
```

References

[1] M. Almgren, H. Debar, and M. Dacier.
A lightweight tool for detecting web server attacks. *In Proceedings of NDSS 2000, Network and Distributed System Security Symposium.* The Internet Society, February 2000.

[2] M. Almgren and U. Lindqvist. Application-Integrated Data Collection for Security Monitoring. *Proceedings of the Fourth International Symposium on the Recent Advances in Intrusion Detection (RAID'2001)*, number 2212 in LNCS, pages 22-36, 2001.

[3] R. Bace and P. Mell.
Intrusion Detection Systems. *NIST Special Publication on Intrusion Detection Systems.* National Institute of Standards and Technology, August, 2001.

[4] T. V. Ryutov and B. C. Neuman.
The Set and Function Approach to Modeling Authorization in Distributed Systems.
In Proceedings of the Workshop on Mathematical Methods and Models and Architecture for Computer Networks Security, May 2001, St. Petersburg Russia.

[5] Sanctum, Inc. *http://www.sanctuminc.com/*

Appendix C:

Dynamic Authorization and Intrusion Response in Distributed Systems

Tatyana Ryutov, Clifford Neuman and Dongho Kim
Information Sciences Institute
University of Southern California
{tryutov, bcn, dongho}@isi.edu

Abstract

This paper[1] presents an authorization framework for supporting fine-grained access control policies enhanced with light-weight intrusion/misuse detectors and response capabilities. The framework intercepts and analyzes access requests and dynamically adjusts security policies to prevent attackers from exploiting application level vulnerabilities.

We present a practical, flexible implementation of the framework based on the Generic Authorization and Access Control API (GAA-API) that provides dynamic authorization and intrusion response capabilities for many applications. To evaluate our approach, we integrated the API with several applications, including Apache web server [12], sshd and FreeS/WAN IPsec for Linux. This paper demonstrates the integration of the GAA-API into ssh daemon. By integrating the GAA-API into sshd, the ssh server can support fine-grained authorization policies, dynamic policy update, and application level intrusion detection and response. The server can also enforce policies with additional functionalities, e.g., time- and location-based controls. Our experiments showed that the required integration effort was moderate, and that the performance impact on the ssh server was negligible.

1 Introduction and Motivation

As more and more enterprises make their critical information available on the Internet, whether only to employees or to customers, they are exposed to significant risks such as theft, fraud, and denial of service attacks. In general, the most significant consequences result from attacks within the system by otherwise legitimate users (or attackers posing as such users) performing unauthorized activities.

[1] Portions reprinted, with permission, from T. V. Ryutov and B. C. Neuman. The Specification and Enforcement of Advanced Security Policies. In the Proceedings of the Conference on Policies for Distributed Systems and Networks (POLICY 2002). ©2002 IEEE.

Detecting these kinds of attacks can require instrumenting applications to generate audit records based on activity that is only understood at the application layer.

Countermeasures to such attacks must similarly be implemented at the application layers through enforcement of policies that can distinguish legitimate and illegitimate activities - a distinction that often requires application level knowledge.

The policies themselves must automatically adapt to meet the changing security requirements in the event of possible intrusion while allowing users to operate in the changing environment.

Access control policies can assist in the application-based category of intrusion detection, which monitors critical applications. Traditional access control policies simply specify whether the access is granted or whether the request is denied. A new policy specification approach with intrusion detection in mind (in addition to defining actions that are and are not permitted) will identify specific application level events that constitute malicious or suspicious activities. Furthermore, such policies will specify the countermeasures to be taken to respond to the suspected or detected attacks.

We apply dynamic authorization techniques to support fine-grained access control and application level intrusion/misuse detection and response capabilities.

Our approach is based on specifying access control policies extended with the capability to identify (and possibly classify) intrusions and respond to the intrusions in real time. The Generic Authorization and Access Control API (GAA-API) is a generic interface which may be used to enable such dynamic authorization and intrusion response capabilities for many applications. The API supports three policy enforcement phases:

1. Before requested operation starts; to decide whether this operation is authorized.

2. During the execution of the authorized operation; to detect malicious behavior in real time (e.g., a user process consumes excessive system resources).

3. When the operation is completed; to activate post execution actions, such as logging and notification whether the operation succeeds/fails.

This paper demonstrates the integration of the GAA-API into ssh daemon. By integrating the GAA-API into sshd, the ssh server can support fine-grained authorization policies, dynamic policy update, and application level intrusion detection and response. The server can also enforce policies with additional functionalities, e.g., time- and location-based controls. Our experiments showed that the required integration effort was moderate, and that the performance impact on the ssh server was negligible with relatively small configuration and policy files.

2 Approach

An *authorization policy* regulates access to objects. An *object* is a target of requests and it has to be protected, e.g., critical programs, files and hosts. An *access right* (alternative words that we use are operation and action) is a particular type of access to a protected object, e.g., read or write. Specific system events, such as restarting or shutting down the system, system log-in and log-off can be modeled as access rights associated with the system, where the system is the protected object. A *condition* describes the context in which each access right is granted or denied.

In our framework, a policy is represented as a set of conditions associated with a positive or negative access right. If all conditions associated with a positive right are met, the access to a target object is granted. If all conditions associated with a negative right are met, the access is denied.

Traditional security systems lack adaptive security policies and enforcement mechanisms. In the non-adaptive setting, the set of policies is chosen in advance, before the access request is received. The adaptive policy enforcement mechanism chooses the appropriate set of policies during the course of computation based on the current system state.

Usually, adaptive policy implementation requires either the reloading of the policy or changing the policy computation algorithms [3]. Both of these approaches are ineffective and not scalable.

Our approach avoids policy reloading and switching to the different policy evaluation mode:

1. The policy specification describes more than one set of disjoint policies.

2. The policy evaluation mechanism is extended with the ability to read and write system state. The implementation is based on conditions that provide support for monitoring and updating internal system structures and their runtime behaviors.

With the extended policy evaluation mechanism, transition between the disjoint sets of policies is regulated automatically by reading the system state (e.g., the time of day, or system threat level). The downside of this approach is the requirement for more tedious and careful policy specification and dealing with the side effects of the policy evaluation.

The adaptive policies are specified using different conditions that permit run-time adaptation in the event of possible security attacks. To enforce the adaptive policies we adopted the three-phase policy enforcement scheme. During each phase only the specified set of all conditions in the policy is evaluated.

2.1 Conditions

Here we list several of the more useful conditions [10] that assist in detecting and responding to intrusion and misuse and they allow more efficient utilization of security services, such as authentication, audit, and notification.

- **access identity**

  This condition specifies an authenticated access identity.

- **strength of authentication**

  This condition specifies the authentication mechanism or set of suitable mechanisms for authentication. Strong user authentication method (e.g., Kerberos [11]) can be activated in response to suspicious behavior.

- **time**

  This condition specifies time periods for which access is granted.

- **location**

  This condition specifies location of the user. Authorization is granted to the users residing on specific hosts, domains, or networks.

- **payment**

  This condition specifies a currency and an amount that must be paid prior to accessing an object.

- **quota**

  This condition specifies a currency and a limit. It limits the quantity of a resource that can be consumed or obtained.

- **audit**

  This condition enables automatic generation of audit data in response to access requests. An audit record should include sufficient information to establish what event occurred and what caused the event.

- **notification**

  This condition enables automatic generation of notification messages (alerts) in response to access requests. The condition value specifies the receiver and the notification method.

- **threshold**

  This condition specifies allowable threshold.

- **system threat level**

  This condition specifies the system threat level.

Failure of some of these conditions may signal suspicious behavior. For example, access is requested at unexpected times or unusual locations, violations of user quotas, repeated failure of access attempts and exceeding a threshold. Some conditions can trigger defensive measures in response to perceived system threat level. For example, impose a limit on resource consumption, advanced payment for the allocated resources or increased auditing. In the case of insider misuse (particularly if the intruder's identity has been established) it may be appropriate to let the attacks continue under special conditions. For example, it may be desirable to initiate data collection mechanisms to gather detailed information about user activities that could serve as evidence for possible prosecutions.

The combination of conditions of different types can be used to fine tune audit and notification services. The audit detail and number of alarms should be sensitive to the system threat profile. For example, low system threat level should result in reduced alarm level and amount of generated audit data. It should also depend on the sensitivity of the requested operation and target object.

2.1.1 Evaluation of Conditions

Note that in the implementation, some of these conditions might have side effects. For example, evaluation of **payment** condition reduces a balance. Evaluation of **notification** condition results in sending a message, which is useful in audit.

A positive aspect of the side effects is the ability to update system behavior at run time (e.g., generating audit records and reconfiguring firewall rules). Such dynamic techniques will ensure that policies applied to system services adapt to perceived system threat profile, thereby increasing system protection.

Unfortunately, side effects complicate matters. There are two particular difficulties in reasoning about policies enforced in the dynamic authorization environment.

First, the side effects might cause problems when the side effects create a feedback loop, e.g., when payment affects quotas which affects the ability to perform other operations (once one runs out of money).

Second, policy rules can include both environmental conditions and actions that change the conditions. For example, an audit condition may trigger a network threat detection condition which affects the evaluation of subsequent conditions in the policy. Therefore, the consistency and correctness of the access control desicions may depend on the condition evaluation order.

In our current framework, the condition evaluation process is totally ordered. The order has to be assessed before condition evaluation starts. Determining the evaluation ordering is currently done by a policy officer.

We recognize that the function of defining the condition order can be best served by an automated tool to ensure policy correctness and consistency and to ease the policy specification burden on the policy officer. See section 8 for further discussion.

2.1.2 Pre-, Mid-, Post- and Request-result Conditions

An authorization policy may specify conditions that must be satisfied before, during or after the access right is exercised. Furthermore, evaluation of some conditions must be activated whether the access is granted or whether the request is denied. Thus, all conditions are classified as:

- **pre-conditions** specify what must be true in order to grant or deny the request.

- **request-result conditions** must be activated whether the access request is granted or whether the request is denied.

- **mid-conditions** specify what must be true during the execution of the requested operation. The mid-conditions can be used for the protection of the critical operations and resources. The mid-conditions allow for real time active monitoring of the operation execution and response. If any of the mid-conditions fails, the operation execution must be affected. The countermeasures are defined in the response methods of the target object. Aggressive responses may include direct countermeasures, such as closing the connections or suspending the processes. This is important to enforce counter measures against serious attacks. For example, a processes consuming excessive system resources (CPU time, memory, and disk space) may indicate impending denial of service attack. More passive responses may include the activating of integrity-checking routines to verify the operating state of the target.

  The mid-conditions that we consider in our framework are limited to a set of thresholds, such as duration of connection, CPU and memory usage and severity metrics (e.g., current system threat level).

- **post-conditions** are used to activate post execution actions, such as logging and notification whether the operation succeeds or whether the operation fails. The post-conditions can be specified in two ways:

    1. The post-conditions that are activated only if the requested operation succeeds. These conditions are useful to correctly implement the enforcement of, for example, the payment/quota constraints.

       Here are some examples of the policies with post-conditions:

       "A user must pay $1 to read a file. The money must be withdrawn from the user account only after successful file access."

       In this policy, the **payment** condition must be implemented as a post-condition. If the file read fails for technical reasons (the server crashes in the middle of the read operation), the payment condition is not activated and the user does not lose his money.

       "A user is allowed to access file A only once."

       Similarly, the **quota** condition in this policy must be implemented as a post-condition to ensure that the user can access the file at least once.

    2. The post-conditions that are activated only if the requested operation fails. For example, failure of critical operations, such as system shut down may indicate denial of service attack and require immediate notification.

The post-conditions along with the request-result conditions are useful to fine tune audit and notification services.

2.2 The Three-Phase Policy Enforcement

The enforcement of the adaptive security policies is partitioned into three successive phases.

1. Phase one: access control.
   The pre- and request-result conditions are evaluated during this phase and the decision to grant or deny access to the requested object is made.

2. Phase two: execution control.
   The access to the target object is granted, the requested operation is started and the mid-conditions are evaluated during this phase. This phase allows the controlled execution of the requested operation.

3. Phase three: post-execution actions.
   The post-conditions are evaluated during this phase. The specified actions are performed after the operation is finished. We do not call this phase "post-execution control", since neither failure nor success of a post-execution action can affect either access decision, or operation execution.

3 Implementation

In this section we present the overview of our implementation approach.

3.1 Policy Representation

The policy language we implemented is called Extended Access Control List (EACL). The EACL is a simple language designed to describe user-level security policies that govern access to protected resources, identify threats that may occur within application and specify intrusion response actions. An EACL is associated with an object (or a group of objects) to be protected and specifies positive and negative access rights with optional set of associated conditions that describe the context in which each access right is granted or denied.

A **condition block** defines a conjunction of a totally ordered set of conditions. Conditions are evaluated in the order they appear within a condition block[2].

An **EACL entry** consists of a positive or negative access right and four condition blocks: a set of pre-conditions, a set of request-result conditions, a set of mid-conditions and a set of post-conditions. Note that a condition block can be empty. If all condition blocks in an EACL entry are empty, the right is granted unconditionally. An example of a practical policy with empty condition blocks is: "anyone can read file $index.html$".

An **EACL** consists of an ordered set of disjunctive EACL entries. An EACL representation supports disjunction and conjunction of conditions to activate different control modes.

An EACL is equivalent to disjunctive normal form consisting of a disjunction of conjunctions where no conjunction contains a disjunction. For example, a policy "Tom or Joe can read file A only if they connect from *.isi.edu domain" can be represented by an EACL (attached to the file A) with two EACL entries:
"positive access right: read, pre-conditions: Tom, *.isi.edu"
"positive access right: read, pre-conditions: Joe, *.isi.edu".

3.1.1 EACL Syntax

We use the Backus-Naur Form to denote the elements of our EACL language. Curly brackets, $\{\}$, surround items that can

[2]The total order property is important to deal with possible side effects caused by the condition evaluation.

repeat zero or more times. A vertical line, |, separates alternatives. Items inside double quotes are the terminal symbols. An EACL is specified according to the following format:

```
eacl ::= {eacl_entry}
eacl_entry ::= pos_access_right conditions |
neg_access_right pre_cond_block
pos_access_right ::= "pos_access_right"
def_auth value
neg_access_right ::= "neg_access_right"
def_auth value
conditions ::= pre_cond_block mid_cond_block
rr_cond_block post_cond_block
pre_cond_block ::= {condition}
mid_cond_block ::= {condition}
rr_cond_block ::= {condition}
post_cond_block ::= {condition}
condition ::= cond_type def_auth value
cond_type ::= alphanumeric_string
def_auth ::= alphanumeric_string
value ::= alphanumeric_string
```

`cond_type` defines the type of condition, e.g., access identity or time.

`def_auth` indicates the authority responsible for defining the value within the `cond_type`, e.g., Kerberos.

`value` is the value of condition. Its semantics is determined by the `cond_type` field. The name space for the value is defined by the `def_auth` field.

Note that the EACL syntax allows only the pre-conditions to be associated with a negative right. This is because an EACL entry with a negative right can never grant the access, therefore, the mid- and post-conditions in the entry will never be evaluated.

We next present an example of an EACL that governs access to a host.

Entry 1 specifies that user tom@ORGB.EDU can not login to the host.

Entries 2 and 3 mean that user Joe can shut down the host using either X509 or Kerberos for authentication. If the request succeeds, the user ID must be logged. If the operation fails, the system administrator must be notified by e-mail.

Entry 4 means that anyone, without authentication, can check the status of the host if he connects from the specified IP address range.

Entry 5 specifies that user ken@ORGA.EDU can login from the specified IP address range, if the number of previous login attempts during the day does not exceed 3. If the request fails, the number of the failed logins for the user must be updated. The connection duration time must not exceed 8 hours.

```
# EACL for host malta.isi.edu
# EACL entry 1
neg_access_right test host_login
pre_cond_access_id_USER Kerberos5
tom@ORGB.EDU

# EACL entry 2
pos_access_right test host
shut_down
pre_cond_access_id_USER X509
"/C=US/O=Trusted/OU=orgb.edu/CN=Joe"
rr_cond_audit local on:success/userID
post_cond_notify local
on:failure/admin/userID

# EACL entry 3
pos_access_right test host_shut_down
pre_cond_access_id_USER Kerberos5
joe@ORGB.EDU
rr_cond_audit local on:success/userID
post_cond_notify local
on:failure/admin/userID

# EACL entry 4
pos_access_right test host_check_status
pre_cond_location IP 10.1.1.0-10.1.2.255

# EACL entry 5
pos_access_right test host_login
pre_cond_access_id_USER Kerberos5
ken@ORGA.EDU
pre_cond_location IP 10.1.1.0-10.1.2.255
pre_cond_threshold local
≤3failures/day/failed_log
rr_cond_update_log local
on:failure/failed_log/userID
mid_cond_duration local ≤8hrs
```

Evaluation of an EACL starts from the first to the last in the list of EACL entries. The resolution of inconsistent authorization is based on ordering. The entries which already have been examined take precedence over new entries.

An ordered evaluation approach is easier to implement as it allows only partial evaluation of an EACL and resolves the authorization conflicts. The problem with this approach is that it requires total ordering among authorizations. It requires careful writing of the EACL by the security administrator.

3.2 Generic Authorization and Access-control API(GAA-API)

The GAA-API provides a general-purpose execution environment in which EACLs are evaluated. We next provide a brief description of the main GAA-API functions.

The $gaa\_get\_object\_policy\_info$ function is called to obtain the security policy associated with the object. It takes the target object and authorization database as input and returns an ordered list of EACLs.

The application maintains authorization information in a form understood by the application. It can be stored in a file, database, directory service or in some other way. The application-specific function provided for the GAA-API retrieves the policy information and translates it into the internal representation understood by the GAA-API. Currently the policy is written at the object level, the call-back function must collect all the per object policies and order them by priority. How the policies are stored and retrieved is opaque to the GAA-API and is not reflected in the EACL.

The resulting policy that is passed to the GAA-API for evaluation represents the combination of several policies possibly from different domains and individual users of the system.

The specific mechanism for retrieving the policies is passed to the GAA-API as a call-back function. The GAA-API provides a mechanism to register a particular policy retrieval call-back function. Currently this is done using a configuration file.

The structure of the policy domains that contribute the policies is not specified explicitly in our framework. Only the hierarchical relationship (priority of the policy) among the domains is taken into consideration. Our current implementation supports two level policy specification: first, system-wide policies are retrieved and placed in the beginning of the list of policies. Then the local policies are retrieved and are added to the list. Thus, system-wide policies implicitly have higher priority than local policies. For further discussion of the policy composition see [12].

The $gaa\_check\_authorization$ function checks whether the requested right is authorized under the specified policy. This function takes the retrieved policy (an ordered list of EACLs), requested access right and contextual information as input. The contextual information is matched to the requirements, specified in the conditions of the relevant EACL entries (only the EACL entries where the the requested right appears are evaluated). This information can be represented by a set of credentials, e.g., an X.509 identity certificate. The output lists all matching policy rights and associated conditions, with flags set to indicate whether each condition was evaluated and/or met. If the access is granted, the output includes the time period for which the result is valid.

The $gaa\_execution\_control$ function performs policy enforcement during operation execution. This function checks whether the mid-conditions associated with the granted access right are met.

The $gaa\_post\_execution\_actions$ function performs policyenforcement after the operation completes. This function enforces the post-conditions associated with the granted access.

An EACL may specify conditions of different types, e.g., **access identity**, **location** and **audit**. The GAA-API supports registering condition evaluation functions for different condition types.

The configuration file lists concrete functions that implement the conditions. The file is read at the GAA-API initialization time and the functions are registered with the specific conditions (taking into account the condition type and defining authority fields). To evaluate conditions in the EACL example given earlier, we might register up to 8 functions [3] with the GAA-API. The GAA-API is structured to support the addition of modules for evaluation of new conditions.

The GAA-API returns three status values to describe policy enforcement process:

1. authorization status S_a.
   Indicates whether the request is authorized (GAA_YES), not authorized (GAA_NO) or uncertain (GAA_MAYBE).

2. mid-condition enforcement status S_m.
   Indicates the evaluation status of the mid-conditions ($GAA\_YES/GAA\_NO/GAA\_MAYBE$).

3. post-condition enforcement status S_p.
   Indicates the evaluation status of the post-conditions ($GAA\_YES/GAA\_NO/GAA\_MAYBE$).

The status values are obtained during the evaluation of conditions in the relevant EACL entries:
GAA_YES - all conditions are met;
GAA_NO - at least one of the conditions fails;
GAA_MAYBE - none of the conditions fails but there is at least one condition that is left unevaluated.

The GAA-API returns GAA_MAYBE if the corresponding condition evaluation function is not registered with the API. In some cases, it is convenient to return some of the conditions unevaluated for further evaluation by the calling application.

4 The GAA/ssh Integration

Secure shell (ssh) is being widely deployed because of its features that ensure secure communications across the net-

[3] Depending on the implementation, we may register either one or two functions to evaluate conditions of the same type but with different defining authority fields, e.g., $pre\_cond\_access\_id\_USERX509$ and $pre\_cond\_access\_id\_USERKerberos5$.

work as well as its ease of use. However, correctly configuring the server (sshd) with desired policies is not an easy task, because the authorization policies are described in the server configuration file that contains not only the policies but also the configuration parameters for the server. Hosting two separate functionalities into one configuration file leads to the problem of having inflexible mechanism of describing authorization policies. In addition, the server has to be restarted after modifying the content of the configuration file to reflect changes. If the modification was done to change the policy, instead of the the configuration of the server, restarting the server would prohibit the dynamic response of the server to the potential or actual network threat conditions.

4.1 The Policy Enforcement Process

The GAA-API was integrated into Openssh version 2.9p2 (http://www.openssh.org). The integration contributed only about 250 lines of “glue” code. Only two files auth2.c and serverloop.c were modified and one new file gaa-plug.c (containing the GAA-API initialization and access control calls) was added.

The GAA-API is integrated into ssh by modifying the $userauth\_reply$ function in the file auth2.c. The file serverloop.c was modified to support the execution control and post-execution phases. The GAA/ssh integration is shown in Figure 1.

The GAA-API makes use of system-wide and local configuration and policy files. The configuration files list routines and parameters for evaluating conditions specified in the policy files. The system-wide policy applies to all applications in the system. The local policy describes security requirements of sshd.

1. The **initialization phase**.

   When the sshd starts, first the GAA-API is initialized by calling $gaa\_initialize$ and gaa_new_sc that extract and register condition evaluation and policy retrieval routines from the system and local configuration files, fetch the system policy file, and generate internal structures for later use.

2. The **access control phase** starts with receiving a connection request.

   (a) The $gaa\_get\_object\_policy\_info$ function is called to obtain the security policies associated with the target host. The function reads the system-wide policy file, converts it to the internal EACL representation and places it at the beginning of the list of EACLs. Next, the function retrieves and translates the local policy file and adds it to the list.

   (b) The request is converted into a list of requested access rights. The authenticated user identity is extracted from the $authctxt$ structure and is placed in the gaa_sc security context structure.

   (c) Next, the $gaa\_check\_authorization$ function is called to check whether the requested right is authorized by the ordered list of EACLs. This function finds the EACL entries where the the requested right appears and calls the registered routines to evaluate pre- and request-result conditions in the entries. If there are no pre-conditions, the authorization status is set to GAA_YES. Otherwise, the pre-conditions are evaluated and the result is stored in the authorization status S_a.

   If the request-result conditions are present in the entry, the conditions are evaluated and the intermediate result is calculated. The conjunction of the intermediate result and S_a is stored in the authorization status S_a.

   Based on the authorization status S_a the connection is permitted or rejected as follows:
   $S_a = GAA\_YES$ connection is allowed.
   $S_a = GAA\_NO$ connection is rejected.
   $S_a = GAA\_MAYBE$ connection is rejected.
   The detailed information is returned that lists all matching policy rights and associated conditions, with flags set to indicate whether each condition was evaluated and/or met.

3. The **execution control phase** consists of starting the connection and calling the $gaa\_execution\_control$ function. This function checks whether the mid-conditions associated with the granted access rights are met. If mid-conditions are found, the conditions are evaluated. Some mid-conditions are evaluated just once [4], other mid-conditions are evaluated in a loop until either the operation finishes or any of the mid-conditions fails. In the latter case, the operation execution is suspended and the reactive actions are started. The mid-conditions can be returned unevaluated to be enforced by application. The result is stored in S_m.

4. During the **post-execution action phase** the $gaa\_post\_execution\_actions$ function is called to enforce the post-conditions associated with the granted rights. This function performs policy enforcement after the operation completes by executing actions such as notifying by email, modifying system variables, writing log file, etc.

   The connection status (indicating whether the connection succeeded/failed) is passed to the

[4] E.g., locking a file to place a hold on user account.

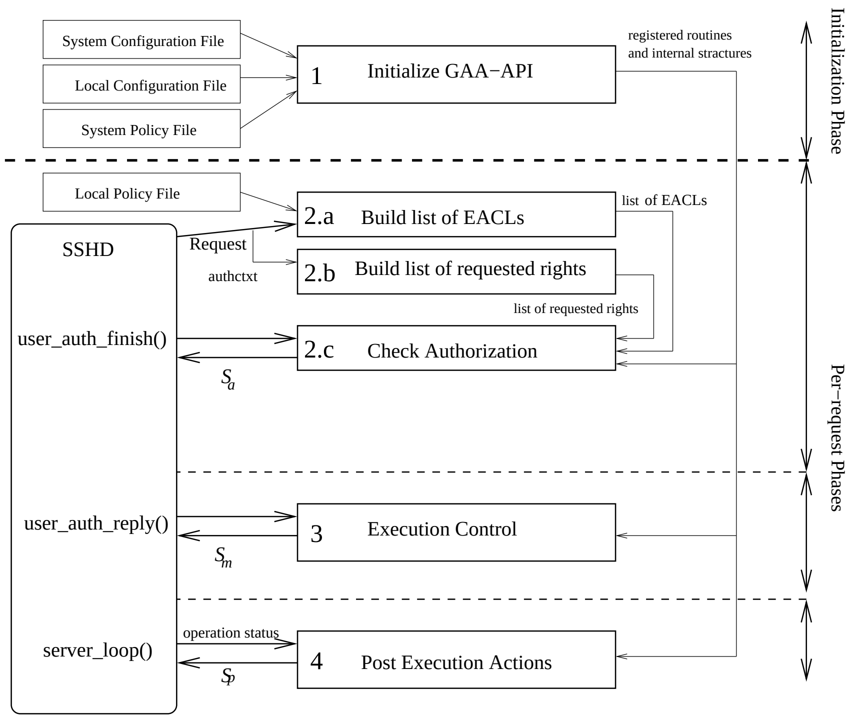

Figure 1. GAA/ssh integration.

gaa_post_execution_actions function. If no post-conditions are found, GAA_YES is returned, otherwise the post-conditions are evaluated and the result is returned in S_p.

5 Deployments

In this section we illustrate how our framework can be deployed to enable fine-grained response to attacks.

5.1 Network Lockdown

This scenario demonstrates how our system adapts the applied authentication policies to require more information from a user when potentially dangerous activity has been detected.

This scenario is designed for organizations with the following characteristics:

- Mixed access to web services. Access to some web resources require user authentication, some do not. If a policy does not require authenticated user identity, authentication steps can be ignored or deferred until the policy explicitly requests it. An example of a policy, which is not concerned with the identity is ”anyone can read file A if $10 is paid”.
- Authenticated ssh connections from the Internet are allowed to access hosts on the organization’s LAN.
- A network-based IDS supplies a system threat level. For example, *low* threat level means normal system operational state, *medium* threat level indicates suspicious behavior and *high* threat level means that the system is under attack.
- Policy: **When system threat level is higher than *low*, one needs to lock down the system and require user authentication for all accesses within the LAN.** Strong authentication protects against outside intruders. To some extent, authentication may help to reduce insider misuse. In particular, insiders are discouraged if the identity of a user can be established reliably.

The policy requirements can be represented by the following EACL that protects all ssh and web server connections within the LAN:

```
# EACL entry 1
pos_access_righ apache *
pre_cond_system_threat_level local >low
pre_cond_access_id_USER apache *

# EACL entry 2
pos_access_righ ssh *
pre_cond_system_threat_level local >low
pre_cond_access_id_USER ssh *
```

The pre-conditions in EACL entries 1 and 2 mean that all Apache and ssh accesses have to be authenticated if the system threat level is higher than *low*. Currently the implementation of the `pre_cond_system_threat_level` condition retrievs system threat level from a specific file.

5.2 Application Level Intrusion Detection

We next demonstrate how our framework provides real time application level intrusion/misuse detection capabilities. This example demonstrates detection and response to a particular DoS attack: opening a large number of simultaneous connections to the ssh server starves the number of available sockets, disallowing new connects.

Assume that EACLs that govern hosts within the LAN contain the following EACL entry:

```
pos_access_right ssh host_login
pre_cond_access_id_USER X509 *
pre_cond_threshold local ≤20/user_sessions
rr_cond_notify local
on:failure/admin/ssh,DoS
rr_cond_update_log local
on:failure/failed_log/userID
mid_cond_update_log local
user_sessions/userID+1
post_cond_update_log local
on:success/user_sessions/userID-1
```

Evaluation of the `pre_cond_access_id_USER` asserts a proper user authentication. The pre-condition `pre_cond_threshold` reads the log of active sessions to determine the number of sessions with the user ID field equal to the one in the user ID credentials.

If the number is greater than 20, the request is rejected. The `rr_cond_notify` condition sends e-mail to the system administrator reporting time, user name and a threat type. Next, the `rr_cond_update_log` updates the log of failed logins to include a new suspicious user ID. If the number of such sessions is less than 20, the request is granted, the connection is established and the mid-condition `mid_cond_update_log` is evaluated. This condition is evaluated just once, it updates the number of active ssh connections for the user. After a connection is closed, the post-condition `post_cond_update_log` updates the number of connections reducing it by 1.

6 Performance

7 Related Work

The Policy Maker system described in the papers by Blaze et al. [1], [2] focuses on construction of a practical algorithm for a determining trust decisions. Policies and credentials encode trust relationships among the issuing sources.

In Policy Maker's terminology, ”proof of compliance question” asks if the request q, supported by a set of credentials complies with a policy P. This is equivalent to the authorization question that we consider in our work: ”is request q authorized by the policy P (in our model the credentials are contained in the request)”. Their approach, however, is different from ours.

In our approach, the information passed to the authorization engine with the authorization request is used to evaluate conditions in the relevant policy statements. The order of condition evaluation is important.

The Policy Maker system is based on the logic programming approach. The goal is to infer the desired conclusion from given assumptions in a computationally viable manner. In Policy Maker, the credentials and policy (called assertions) are used collectively to compute a proof of compliance. The assertions can be run in an arbitrary order and produce intermediate results that then can be fed into other assertions.

Hayton and colleagues [5] proposed a role-based access control system called OASIS. OASIS services specify policy for role activation using Role Definition Language (RDL) that is defined in terms of axioms in proof system. These axioms are used to prove user's eligibility to enter a set of roles.

A role can be specified as being permitted only for those who can prove membership of other roles issued by this and other services. The services are responsible for issuing certificates, verifying their validity and notifying other services about the certificate state changes. A policy defines a set of conditions under which a user can activate a role. The role revocation is accomplished through membership conditions. Some of the membership conditions must continue to hold while the role remains active. If any of the membership conditions associated with the activated role fails, the role is deactivated. In some sense, the OASIS membership conditions are similar to our mid-conditions that must hold during operation execution.

RDL is not as generic and expressive as our approach and not as well suited to representing complex access control policies and those that include mandatory access control.

Policies, representable in Policy Maker and RDL, are restricted to the set of policies which do not produce side effects, resulting in change of the system state.

Ponder [4] is an object-oriented policy specification language that is suted for role-based access control policies, as well as general-purpose management policies. Ponder is targeted for different types of policies, including obligations, authorizations, delegation and filtering policies, and grouping these policies into aggregate structures. The obligation policies, for example, specify what actions (e.g., notification or logging) are carried out when specific events occur within the system. To some extent, the request-result and post-conditions in our framework serve a similar purpose. However, there are several significant differences between Ponder's and our approaches. First, in our framework all security requirements are expressed in a single policy structure, whereas in the Ponder approach authorization and obligation policies can be specified independently. These can lead to conflicts between the two policy types. Second, the policy in our framework is enforced by the same access control mechanism. The three-phase policy enforcement model allows for parts of policy (particular conditions) to be enforced at different times. In contrast, the Ponder uses a separate enforcement mechanism for each policy type.

Finally, the Ponder obligation policies are triggered by system events whereas in our framework the actions are triggered by other conditions in the same policy, such as threshold or system threat level.

Minsky and Ungureanu [8], [9] define the policy in terms of messages that only a restricted set of agents is permitted to exchange. Furthermore, the message exchange is controlled by a set of rules that is included in the policy. The policy enforcement mechanism is based on a set of trusted agents that interpret the rules and enforce them by regulating the message exchanges and the effect that the messages have on the control state (attributes and permissions) of the participating agents.

The ability to communicate and change the state resembles our concept of the read and write conditions. Our approach is different in that the "state" has a wider meaning. It includes all security-relevant information about real world which is representable in a computer system, e.g., bank account balance, temperature and user identity. Another difference is that the reading and writing of the state is based on the ordered synchronous evaluation of the conditions, rather than controlled message exchange.

Jajodia et al. [6] have proposed a logical language for the specification of authorizations. The concerns addressed in this work are orthogonal to the ones in this paper. In particular, they focus on modeling conflict resolution, integrity constraint checking and derivation rules (that derive implicit authorizations from explicit ones), while our work focuses on the representation and enforcement of authorization policies enhanced with detection and management of security violations.

Summary of the research of audit-based intrusion and misuse detection is given by Lunt [7]. Sandhu and Samarati [13] discuss authentication, access control and intrusion detection technologies and suggest that combination of the techniques is necessary in order to build a secure system.

8 Conclusions and Future Work

Traditional authorization mechanisms check whether a user is acting within prescribed parameters and will not detect abuse of privileges. In this paper we presented an authorization framework that enables the specification and enforcement of workable security policies that govern access to protected resources, identify threats that may occur within application and specify intrusion response actions. The GAA-API combines policy enforcement with application level intrusion detection and response, allowing countermeasures to be applied to ongoing attacks before they cause damage. The GAA-API implementation is available at **http://www.isi.edu/gost/info/gaaapi/source**.
The GAA-API has been integrated with several applications, including Apache web server [12], sshd and FreeS/WAN IPsec for Linux.

Currently, the GAA-API integrated sshd obtains part of the policy from the original sshd configuration file (to maintain the backward compatibility) and uses the policy file specifed in EACL format to supplement the existing policy. We plan to improve the GAA/ssh integration to completely take over the authorization phase of sshd.

In the current framework, we assume that conditions are evaluated consecutively and that authorization requests do not overlap. These two assumptions enable us to concentrate on a single condition evaluation per each time interval and, therefore, avoid the problem of coordination of multiple condition evaluation processes. However, this approach results in inefficient policy evaluation process and leads to systems that cannot scale to large numbers of objects.

The future directions for this research include exploring extensions to the framework to support: concurrent requests; replication of the evaluation mechanism; concurrent evaluation of conditions within the same request; and distributed policy enforcement.

At this point, the issues of spatial and temporal relationships among the policy computations become critical. Policies that govern the same object may have non-trivial interdependencies which must be carefully analyzed and understood.

Another limitation of the current framework is reliance on a policy administrator for defining condition evaluation order which is then enforeced by the framework. The limited awareness of the spatial and temporal dependencies among security policies may cause inconsistencies and undesirable system behavior. In many cases, administrators may not have a clear picture of the ramifications of policy enforcement actions, therfore enforcing these policies might have unexpected interactive or concurrent behaviour. Automation is essential to minimise human error, and it can only be used safely when there is a formal model that explicitly addresses both the spatial and the temporal aspects of dynamic authorization.

We aim to develop a formal model which can be used to create policies with strong security guarantees, eliminating guesswork in the design and deployment of adaptive security policies.

9 Acknowledgments

Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, and the Xerox Corporation under the following agreements: (1) F30602-00-2-0595, Dynamic Policy Evaluation of Containing Network Attacks Project (DEFCN); (2) DABT63-94-C-0034, Security Infrastructure for Large Distributed Systems Project (SILDS); (3) J-FBI-95-204, Global Operating System Technologies Project (GOST); (4) DE-FC03-99ER25397, Diplomat project; and (5) HE1254-97, XAUTH Project.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, U.S. Department of Energy, the U.S. Government or the Xerox Corporation. Figures and descriptions were provided by the authors and are used with permission.

References

[1] M. Blaze, J. Feigenbaum and J. Lacy.
Decentralized Trust Management.
Proceedings IEEE Symposium on Security and Privacy, IEEE Computer Press, Los Angeles, pages 164-173, 1996.

[2] M. Blaze, J. Feigenbaum and M. Strauss.
Compliance Checking in the Policy Maker Trust Management System. *In Proceedings of the Financial Cryptography '98, Lecture Notes in Computer Science*, volume 1465, pages 254-274.

[3] M. Carney and B. Loe.
A Comparison of Methods for Implementing Adaptive Security Policies. *In Proceedings of the 7th USENIX Security Symposium*, pages 1-14, January, 1998.

[4] N.Damianou, N. Dulay, E. Lupu and M. Sloman.
The Ponder Policy Specification Language. *In Proceedings of the Workshop on Policies for Distributed Systems and Networks*, Springer-Verlag LNCS 1995, pages 18-39, Bristol, UK, January, 2001.

[5] R. J. Hayton, J. M. Bacon and K. Moody.
OASIS: Access Control in an Open, Distributed Environment.
Proceedings of the IEEE Symposium on Security and Privacy, pages 3-14, Oakland, CA, May 1998.

[6] S. Jajodia, P. Samarati and V.S. Subrahmanian.
A logical Language for Expressing Authorizations.
Proceedings of the 1997 IEEE Symposium on Security and Privacy, 1997.

[7] T. F. Lunt.
A Survey of Intrusion Detection Techniques.
Computers and Security, volume 12, pages 405-418, June 1993.

[8] N. Minsky and V. Ungureanu.
Unified Support for Heterogeneous Security Policies in Distributed Systems.
In 7th USENIX Security Symposium, San Antonio, Texas, January 1998.

[9] N. Minsky and V. Ungureanu.
Law-Governed Interaction: A Coordination & Control Mechanism for Heterogeneous Distributed Systems. *In ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol 9, No 3, pages 273-305, July 2000.

[10] B.C. Neuman.
Proxy-based authorization and accounting for distributed systems.
In Proceedings of the 13th International Conference on Distributed Computing Systems, Pittsburgh, May 1993.

[11] B.C. Neuman and T. Ts'o.
Kerberos: An authentication service for computer networks.
IEEE Communications Magazine, pages 33-38, September 1994.

[12] T. V. Ryutov, B. C. Neuman, Li Zhou and Dongho Kim.
Integrated Access Control and Intrusion Detection for Web Servers.
In Proceedings of the 23rd International Conference on Distributed Computing Systems, Providence, Rhode Island, May 2003.

[13] R. Sandhu and P. Samarati.
Authentication, Access Control, and Intrusion Detection.

The Computer Science and Engineering Handbook, pages 1929-1948, CRC Press, 1997.

Appendix D:

The Specification and Enforcement of Advanced Security Policies

Tatyana Ryutov and Clifford Neuman
Information Sciences Institute
University of Southern California
{tryutov, bcn}@isi.edu

Abstract

In a distributed multi-user environment, the security policy must not only specify legitimate user privileges but also aid in the detection of the abuse of the privileges and adapt to perceived system threat conditions.

This paper advocates extending authorization policy evaluation mechanisms with a means for generating audit data allowing immediate notification of suspicious application level activity. It additionally suggests that the evaluation of the policies themselves adapt to perceived network threat conditions, possibly affected by the receipt of such audit data by other processes.

Such advanced policies assist in detecting and responding to intrusion and misuse and they allow more efficient utilization of security services, such as authentication, audit, and notification.

We present an authorization framework, which enables the representation and enforcement of advanced security policies. Our approach is based on expanding the policy evaluation mechanism with the ability to generate real time actions, such as checking the current system threat level and sending a notification.

1 Introduction and Motivation

As more and more enterprises make their critical information available on the Internet, whether only to employees or to end-customers, they are exposed to significant risks such as theft, fraud, and denial of service attacks. In general, the most significant consequences result from attacks within the system by otherwise legitimate users (or attackers posing as such users) performing unauthorized activities.

Detecting these kinds of attacks can require instrumenting applications to generate audit records based on activity that is only understood at the application layer.

In addition to having a means to detect attacks (the role of an intrusion detection system) it is essential to have well defined policies that indicate what to do under perceived attack conditions, or for that matter under suspicion of attack conditions so that data can be gathered to make an actual determination of whether an attack is present.

Countermeasures to such attacks must similarly be implemented at the application layers through enforcement of policies that can distinguish legitimate and illegitimate activities - a distinction that often requires application level knowledge.

While users might not be prevented from using resources to which they have legitimate access, protective measures, such as audit analysis along with the threshold control can be used to examine user actions. Consider an authorization policy: “Members of department D can access the printer P. If the number of print jobs created during the day is higher than 20, activate audit to log time, file and account names”. In this policy the threshold is used to detect suspicious use of resources. An audit log can reveal that an individual is printing far more records than the average user, which could indicate the running of a covert business.

The policies themselves must automatically adapt to meet the changing security requirements in the event of possible intrusion while allowing users to operate in the changing environment. For example, consider authorization policy: “Tom can connect to host $malta.isi.edu$ if the system threat level is low (normal operational state). If the system threat level is medium (indicates suspicious behavior), Tom can connect only from a host within the administrative domain $isi.edu$. The connection duration time should not exceed 2 hours. If the system threat level is high (system is under attack), Tom can not connect.”

Current access control systems are based on the premise that once a user is authorized to perform some operation, the access is granted unconditionally. This practice is not likely to detect the abuse of user privileges. To provide additional level of security checks, close monitoring of authorized actions may be necessary. Policies can be applied to controlling execution of the requested actions.

The points of the policy enforcement may include three time phases:

1. Before requested operation starts; to decide whether

this operation is authorized.

2. During the execution of the authorized operation; to detect malicious behavior in real-time (e.g., a user process consumes excessive system resources).

3. When the operation is completed; to activate post execution actions, such as logging and notification whether the operation succeeds/fails.

To protect sensitive and critical system resources in distributed environments, a system must be capable of supporting **advanced security policies**:

1. The policies must be adaptive [1] to accommodate changes in the security requirements and assist in detecting and responding to intrusion and misuse. To do so, the policies should indicate not only what activities are authorized, but also provide the means to detect abuse of user privileges. In particular, the policy should specify when audit records should be generated and allow for immediate notification.

2. Policy enforcement can be required at various time stages of the requested action. Thus, the policies should indicate when the policy has to be enforced.

The ability to enforce advanced policies has practical importance, for example, in computational Grids [5]. Grids are large-scale distributed computing environments that enable applications to use scientific instruments, computational and information resources that are managed by diverse organizations.

System administrators contributing their resources to a Grid will require assurance that the resources are adequately protected. In a Grid setting, the security requirements include:

1. User authentication.
   Authenticated user identity is used to determine who gains access to local resources [2].

2. Resource usage limits (quotas).
   A site-specific resource allocation policy specifies limits on the computational or storage resources to be consumed, such as CPU load, memory usage and disk space. The limits are taken into account when deciding whether to initiate the requested computation. Monitoring execution of the computation on a particular node must be supported to ensure that the process keeps strictly to the limits imposed by the local policy.

[1] The term “adaptive” in this paper is used to indicate that the security policy to be enforced depends on the current state of the system, e.g., system load, system threat level or time of day (more restrictive organizational policy may be enforced during after hours).

[2] Mutual authentication may be required to prove the server identity to the user.

3. Accounting and payment.
   Owners of the resources may hold users accountable for the consumed resources. Accounting may include gathering information about executed computations and consumed resources. The accounting information can be used in payment models for remote service providers.

4. Audit.
   Audit can provide a means to help accomplish individual accountability and provide data to be analyzed by intrusion or misuse detection systems.

5. Intrusion and misuse detection.
   Grids are vulnerable to a large-scale malicious attacks that could cause disruption of the Grid services. Thus, it is essential for Grids to support detection and automatic response to intrusion attempts.

6. Event notification.
   Tools for intrusion detection and fault tolerance can be driven by event services. Alert-level notification messages permit cooperative responses. For example, notification about a computation that exceeds the quotas can signal ongoing denial of service attack. The adequate preventive measures can be taken if the attack is confirmed.

Authentication, authorization, audit, notification and intrusion detection systems are interrelated and should be used together to support effective system security.

The goal of this work is to design an authorization system that supports the advanced security policies.

2 Approach

An authorization policy regulates access to objects. An object is a target of requests and it has to be protected, e.g., critical programs, files, hosts and print jobs.

An access right (alternative words that we use are operation, action and permission) is a particular type of access to a protected object, e.g., read or write. Specific system events, such as restarting or shutting down the system, system log-in and log-off can be modeled as access rights associated with the system, where the system is the protected object.

A condition describes the context in which each access right is granted.

In our framework, a policy is represented as a set of conditions associated with the access right. All conditions must be satisfied in order to allow an operation to be performed on a target object [3].

[3] Our framework supports negative rights. If all conditions associated with the negative right are met, the access is denied.

Traditional security systems lack adaptive security policies and enforcement mechanisms. In the non-adaptive setting, the set of policies is chosen in advance, before the access request is received. The adaptive policy enforcement mechanism chooses the appropriate set of policies during the course of computation based on the current system state.

Adaptive policy implementation requires either the reloading of the policy or changing the policy computation algorithms [3]. Both of these approaches are ineffective and not scalable.

Our approach avoids policy reloading and switching to the different policy evaluation mode:

1. The policy specification describes more than one set of disjoint policies.

2. The policy evaluation mechanism is extended with the ability to read and write system state. The implementation is based on read and write conditions that provide support for monitoring and updating internal system structures and their runtime behaviors.

With the extended policy evaluation mechanism, transition between the disjoint sets of policies is regulated automatically by reading the system state (e.g., the time of day, or system threat level). The downside of this approach is the requirement for more tedious and careful policy specification and dealing with the side effects of the policy evaluation.

The advanced policies are specified using different conditions that permit run-time adaptation in the event of possible security attacks. To enforce the advanced security policies we adopted the three-phase policy enforcement scheme. During each phase only the specified set of all conditions in the policy is evaluated.

2.1 Conditions

Here we list several of the more useful conditions [13] that assist in detecting and responding to intrusion and misuse and they allow more efficient utilization of security services, such as authentication, audit, and notification.

- **access identity**

  This condition specifies an authenticated access identity. If a policy does not require authenticated user identity, authentication steps can be ignored or deferred until the policy explicitly requests it. An example of a policy, which is not concerned with the identity is ”anyone can read file A if $10 is paid”.

- **strength of authentication**

  This condition specifies the authentication mechanism or set of suitable mechanisms for authentication. Strong user authentication method (e.g., Kerberos [14]) can be activated in response to suspicious behavior.

- **time**

  This condition specifies time periods for which access is granted.

- **location**

  This condition specifies location of the user. Authorization is granted to the users residing on specific hosts, domains, or networks.

- **payment**

  This condition specifies a currency and an amount that must be paid prior to accessing an object.

- **quota**

  This condition specifies a currency and a limit. It limits the quantity of a resource that can be consumed or obtained.

- **audit**

  This condition enables automatic generation of audit data in response to access requests. An audit record should include sufficient information to establish what event occurred and what caused the event.

- **notification**

  This condition enables automatic generation of notification messages (alerts) in response to access requests. Specifies the receiver and the notification method.

- **threshold**

  This condition specifies allowable threshold.

- **system threat level**

  This condition specifies the system threat level.

Failure of some of these conditions may signal suspicious behavior. For example, access is requested at unexpected times or unusual locations, violations of user quotas, repeated failure of access attempts and exceeding a threshold. Some conditions can trigger defensive measures in response to perceived system threat level. For example, impose a limit on resource consumption, advanced payment for the allocated resources or increased auditing. In the case of insider misuse (particularly if the intruder’s identity has been established) it may be appropriate to let the attacks continue under special conditions. For example, it may be desirable to initiate data collection mechanisms to gather detailed information about user activities that could serve as evidence for possible prosecutions.

The combination of conditions of different types can be used to fine tune audit and notification services. The audit detail and number of alarms should be sensitive to the system threat profile. For example, low system threat level should result in reduced alarm level and amount of generated audit data. It should also depend on the sensitivity of the requested operation and target object.

2.1.1 Evaluation of Conditions

Note that in the implementation, some of these conditions might have side effects. For example, evaluation of **payment** and **quota** conditions reduce a balance. Evaluation of **notification** condition results in sending a message, which is useful in audit.

Unfortunately, side effects complicate the system. Ignoring the side effects might cause problems when the side effects create a feedback loop, for example, when an audit record triggers a network threat detection which affects the evaluation of subsequent policies, or where payment affects quotas which affects the ability to perform other operations (once one runs out of money).

Another problem caused by the side effects, is possible inconsistency of the authorization result. For example, consider a policy "Tom can shut down host H only if a notification is sent (`notification`) and system threat level is low (`system_threat_level:low`)". Assume that the current system threat level is low. Assume that the notification about Tom shutting down the host triggers high system threat level (this may indicate attempted denial of service attack). There are two ways to evaluate the conditions: first `system_threat_level:low` then `notification`. This evaluation order results in access grant. Another way is to evaluate `notification` condition first then `system_threat_level:low`. This evaluation order results in the denial of the access.

All side effects of the condition evaluation are recorded in the corresponding system variables. At the lowest level, a system variable is an abstraction for bits or bytes in the system that change as the result of system execution. For example, to model a system variable affected by the evaluation of the **notification** condition (a message must be sent), we need better level of abstraction. Thus, a system variable is an abstract notion of a system entity that represents a data item, e.g., a file, a message or a record in a database. Each system variable has a name and a value.

We assume that there exists a set of software components S. Each software component s $(s \in S)$ can access system variables of particular type. For example, a system variable, which represents a file is accessed by a file system. A system variable, which represents a notification is accessed by a notification protocol, or a transport protocol, such as e-mail or http.

We assume that each software component s has abstract $Read$ and $Write$ operations as a part of its functionality. The read operation $s.Read(X)$ returns the value of the system variable X. The write operation $s.Write(X, v\_new)$ assigns a new value v_new to the system variable X.

2.1.2 Read and Write Conditions

At the conceptual level, all conditions can be categorized as:

- Conditions that require reading some system variable and comparing it with the information specified in the policy. For example, evaluation of the **time** condition requires obtaining current time and checking if it fits into the time interval specified in the policy. We call this category of conditions **read conditions**. A read condition is represented as $X op P$, where X is the name of a system variable, P is a constant and op is the operation (e.g., $=$, $\neq$, $<$, $>$) to be performed on the value of the system variable X and the constant P. In implementation, this value maybe either obtained from the request or read using the $s.Read(X)$ operation during the condition evaluation.

- Conditions that require writing some information (e.g., audit) or initiating some action (e.g., notification). We call this category of conditions **write conditions**. A write condition is represented as $X new\_value$, where X is the name of the system variable and new_value is the new value to be assigned.

An obvious relationship between the read and write conditions is if one condition requires reading of a system variable, which is written by the other condition. In our framework, the condition evaluation process is totally ordered. The order has to be assessed before condition evaluation starts. Determining the correct order of the conditions in the policy statement is an important issue. Human judgment is a necessary component in this process. We feel that the function of defining the condition order can be best served by having the policy officer chose a meaningful condition order. In particular, whether the **write conditions** must be evaluated before the **read conditions**. The goal of the system is to faithfully implement the given organizational security policy.

2.1.3 Pre-, Mid-, Post- and Request-result Conditions

An authorization policy may specify conditions that must be satisfied before, during or after the access right is exercised. Furthermore, evaluation of some conditions must be activated only if the authorization request is granted (or denied).

Thus, all conditions are classified as:

- **pre-conditions** specify what must be true in order to grant the request. This means that the requested operation is allowed to be executed on the target object. If any of the pre-conditions fails, authorization is denied.
- **request-result conditions**
  These conditions must be activated whether the authorization request is granted or whether the request is denied.
- **mid-conditions** specify what must be true during the execution of the requested operation. The mid-conditions can be used for the protection of the critical operations and resources. The mid-conditions allow for real time active monitoring of the operation execution and response. If any of the mid-conditions fails, the operation execution must be affected. The countermeasures are defined in the response methods of the target object. Aggressive responses may include direct countermeasures, such as closing the connections or suspending the processes. This is important to enforce counter measures against serious attacks. For example, a processes consuming excessive system resources (CPU time, memory, and disk space) may indicate impending denial of service attack. More passive responses may include the activating of integrity-checking routines to verify the operating state of the target.

  The mid-conditions that we consider in our framework are limited to a set of thresholds, such as duration of connection, CPU and memory usage and severity metrics (e.g., current system threat level).
- **post-conditions** specify what must be true on the completion of the operation execution. The post-conditions can be specified in two ways:
  1. The post-conditions that are activated only if the requested operation succeeds. These conditions are useful to correctly implement the enforcement of, for example, the payment/quota constraints.

     Here are some examples of the policies with post-conditions:

     "A user must pay $1 to read a file. The money must be withdrawn from the user account only after successful file access."

     In this policy, the **payment** condition must be implemented as a post-condition. If the file read fails for technical reasons (the server crashes in the middle of the read operation), the payment condition is not activated and the user does not lose his money.

     "A user is allowed to access file A only once."

     Similarly, the **quota** condition in this policy must be implemented as a post-condition to ensure that the user can access the file at least once.
  2. The post-conditions that are activated only if the requested operation fails. For example, failure of critical operations, such as system shut down may indicate denial of service attack and require immediate notification.

The post-conditions along with the request-result conditions are useful to fine tune audit and notification services.

2.2 The Three-Phase Policy Enforcement

The enforcement of the advanced security policies is partitioned into three successive phases.

1. Phase one: access control.
   The pre- and request-result conditions are evaluated during this phase and the decision to grant or deny access to the requested object is made.
2. Phase two: execution control.
   The access to the target object is granted, the requested operation is started and the mid-conditions are evaluated during this phase. This phase allows the controlled execution of the requested operation.
3. Phase three: post-execution actions.
   The post-conditions are evaluated during this phase. The specified actions are performed after the operation is finished. We do not call this phase "post-execution control", since neither failure nor success of a post-execution action can affect either access decision, or operation execution.

3 Implementation

In this section we present the overview of our implementation approach.

3.1 Policy Representation

The policy language that we implemented is called Extended Access Control List (EACL). The EACL is a simple policy language designed to describe user-level authorization policy. An EACL is associated with an object (or a group of objects) to be protected and specifies positive and negative access rights with optional set of associated conditions.

A **condition block** defines a conjunction of a totally ordered set of conditions. Conditions are evaluated in the order they appear within a condition block[4].

An **EACL entry** consists of a positive or negative access right and four condition blocks: a set of pre-conditions, a set of request-result conditions, a set of mid-conditions and a set of post-conditions. Note that a condition block can be empty. If all condition blocks in an EACL entry are empty, the right is granted unconditionally. An example of a practical policy with empty condition blocks is: "anyone can read file $index.html$".

An **EACL** consists of an ordered set of disjunctive EACL entries. An EACL representation supports disjunction and conjunction of conditions to activate different control modes.

An EACL is equivalent to disjunctive normal form consisting of a disjunction of conjunctions where no conjunction contains a disjunction. For example, a policy "Tom or Joe can read file A only if they connect from *.isi.edu domain" can be represented by an EACL (attached to the file A) with two EACL entries:
"positive access right: read, pre-conditions: Tom, *.isi.edu"
"positive access right: read, pre-conditions: Joe, *.isi.edu".

More precise EACL syntax and an example are given in the Appendix.

Evaluation of an EACL starts from the first to the last in the list of EACL entries. The resolution of inconsistent authorization is based on ordering. The authorizations which already have been examined take precedence over new authorizations.

An ordered evaluation approach is easier to implement as it allows only partial evaluation of an EACL and resolves the authorization conflicts. The problem with this approach is that it requires total ordering among authorizations. It requires careful writing of the EACL by the security administrator.

3.2 Generic Authorization and Access-control API(GAA-API)

The GAA-API provides a general-purpose execution environment in which EACLs are evaluated. Next we provide a brief description of the main GAA-API functions.

The $gaa\_get\_object\_policy\_info$ function is called to obtain the security policy associated with the object. It takes the target object and authorization database as input and returns an ordered list of EACLs.

The application maintains authorization information in a form understood by the application. It can be stored in a file, database, and directory service or in some other way. The application-specific callback function provided for the GAA-API retrieves the policy information and translates it into the internal representation understood by the GAA-API. Currently the policy is written at the object level, the call-back function must collect all the per object policies and order them by priority. How the policies are stored and retrieved is opaque to the GAA-API and is not reflected in the EACL.

The resulting policy that is passed to the GAA-API for evaluation represents the combination of several policies possibly from different domains and individual users of the system. The specific mechanism for retrieving the policies is passed as a call-back function.

The GAA-API provides a mechanism to register a particular policy retrieval call-back function. Currently this is done using a configuration file.

The structure of the policy domains that contribute the policies is not specified explicitly in our framework. Only the hierarchical relationship (priority of the policy) among the domains is taken into consideration. Our current implementation supports two level policy specification: first, system-wide policies are retrieved and placed in the beginning of the list of policies. Then the local policies are retrieved and are added to the list. Thus, system-wide policies implicitly have higher priority than local policies.

The $gaa\_check\_authorization$ function checks whether the requested right is authorized under the specified policy. This function takes the retrieved policy (an ordered list of EACLs), requested access right and contextual information as input. The contextual information is matched to the requirements, specified in the conditions of the relevant EACL entries (only the EACL entries where the the requested right appears are evaluated). For example, this information can be represented by a set of credentials, e.g., an X.509 identity certificate. The output lists all matching policy rights and associated conditions, with flags set to indicate whether each condition was evaluated and/or met. If the access is granted, the output includes the time period for which the result is valid.

$gaa\_execution\_control$ performs policy enforcement during operation execution. This function checks whether the mid-conditions associated with the granted access right are met.

$gaa\_post\_execution\_actions$ performs policy enforcement after the operation completes. This function enforces the post-conditions associated with the granted access.

A policy statement may specify several conditions of different types. For example: "Tom can read file A only between 9am and 6pm". This policy defines two pre-conditions: **access identity** and **time**. Both conditions are **read conditions** (there are two system variables to be read: user access identity and current time).

The GAA-API supports registering condition evaluation functions for different condition types.

[4] The total order property is important to deal with possible side effects caused by the condition evaluation.

The configuration file lists concrete functions that implement the conditions. The file is read at the GAA-API initialization time and the functions are registered with the specific conditions. In our policy example we define two functions: one to check the access identity and the other one to check the time. The read vs. write distinction shows up implicitly in the condition type. A condition evaluation function registered with a condition type knows whether the condition is **read** or **write**. It then parses the condition value and calls the concrete functions that implement the abstract $Read$ and $Write$ operations described in Section 2.1.1. The system variables manipulated by the $Read$ and $Write$ operations, as well as the operations themselves can be ether local or remote. However, our framework requires that the $Read$ and $Write$ operations must be implemented as atomic actions. The GAA-API is structured to support the addition of modules for evaluation of new conditions.

The $gaa\_check\_authorization$, $gaa\_execution\_control$ and $gaa\_post\_execution\_actions$ functions return the evaluation status $T/F/U$.

This status is obtained during the evaluation of conditions in the relevant EACL entries:
T indicates that all conditions are met;
F indicates that at least one of the conditions fails;
U indicates that none of the conditions fails but there is at least one condition that is left unevaluated.

Uncertainty U is introduced into our framework by lack of adequate information to evaluate the condition. For example, a condition may depend on an event that has yet to happen. This means that the value of the system variable returned by the implementation of the abstract $s.Read(X)$ operation is undefined. Another source of uncertainty is inability to find the corresponding condition evaluation function, for example if the function ($s.Read(X)$ or $s.Write(X, v\_new)$) is not implemented or not registered with the GAA-API. Sometimes, it is convenient to return some of the conditions unevaluated for further evaluation by the calling application.

3.3 The Policy Enforcement Process

The GAA-API returns three status values to describe policy enforcement process:

1. authorization status S_a.
   Indicates whether the request is authorized (T), not authorized (F) or uncertain (U).

2. mid-condition enforcement status S_m.
   Indicates the evaluation status of the mid-conditions ($T/F/U$).

3. post-condition enforcement status S_p.
   Indicates the evaluation status of the post-conditions ($T/F/U$).

Initially the status values are set to U.

1. The access control phase starts with receiving a request to access an object, requested type of access and contextual information.

2. First, the $gaa\_get\_object\_policy\_info$ function is called to obtain the security policy associated with the object. If no relevant policy was found, the authorization status is set to F and the request is rejected.

   Next the $gaa\_check\_authorization$ function is called to evaluate pre- and request-result conditions. If there are no pre-conditions (this means that the requested right is granted unconditionally), the authorization status is set to T. Otherwise, the pre-conditions are evaluated and the result is stored in the authorization status S_a.

   If the request-result conditions are present in the policy, the conditions are evaluated and the intermediate result is stored in variable X. The conjunction of the X and S_a is stored in the authorization status S_a. If authorization is not granted ($S_a \neq T$), the request is rejected.

3. The execution control phase consists of starting the operation execution process and calling the $gaa\_execution\_control$ function.

   If mid-conditions are found, the conditions are evaluated. Some mid-conditions are evaluated just once [5], other mid-conditions are evaluated in a loop until either the operation finishes or any of the mid-conditions fails. In the latter case, the operation execution is suspended and the reactive actions are started. The mid-conditions can be returned unevaluated to be enforced by application. The result is stored in S_m.

4. During the post-execution action phase the $gaa\_post\_execution\_actions$ function is called. The operation execution status (indicating whether the operation succeeded/failed) is passed to the $gaa\_post\_execution\_actions$. If no post-conditions are found, the S_p is set to T, otherwise the post-conditions are evaluated and the result is stored in S_p.

4 Related Work

The work by Huang and Shan [8] describes a SQL-like policy definition language. The policy enforcement process allows refining of the initial authorization request (request enhancement) and suggesting alternatives (request rewriting) if the requested resource is unavailable. These actions are performed by the policy enforcement mechanism before

[5] E.g., locking a file to place a hold on user account.

submitting the actual resource retrieval request to the resource manager. This approach is different from ours in that:

1. The point of the policy enforcement is at the creation of the resource request (based on the enhancement/rewriting of the initial request), which complies with existing policies. Then the resource is retrieved without any further checks. In our framework, the request is checked against the policies and is denied/granted or uncertain. No request modifications exist.

2. The approach has a limited condition representation model that does not support side effects.

The Policy Maker system described in the papers by Blaze et al. [1], [2] focuses on construction of a practical algorithm for a determining trust decisions. Policies and credentials encode trust relationships among the issuing sources.

In Policy Maker's terminology, "proof of compliance question" asks if the request q, supported by a set of credentials complies with a policy P. This is equivalent to the authorization question that we consider in our work: "is request q authorized by the policy P (in our model the credentials are contained in the request)". Their approach, however, is different from ours.

In our approach, the information passed to the authorization engine with the authorization request is used to evaluate conditions in the relevant policy statements. The order of condition evaluation is important.

The Policy Maker system is based on the logic programming approach. The goal is to infer the desired conclusion from given assumptions in a computationally viable manner. In Policy Maker, the credentials and policy (called assertions) are used collectively to compute a proof of compliance. The assertions can be run in an arbitrary order and produce intermediate results that then can be fed into other assertions.

Hayton and colleagues [7] proposed a role-based access control system called OASIS. OASIS services specify policy for role activation using Role Definition Language (RDL) that is defined in terms of axioms in proof system. These axioms are used to prove user's eligibility to enter a set of roles.

The policy for each set of services is specified at administrative domain level, with service level agreements between domains. The role names are local to each service. A role can be specified as being permitted only for those who can prove membership of other roles issued by this and other services. The services are responsible for issuing certificates, verifying their validity and notifying other services about the certificate state changes. A policy defines a set of conditions under which a user can activate a role. Condition evaluation is achieved by presenting a corresponding certificate. The role revocation is accomplished through membership conditions. Some of the membership conditions must continue to hold while the role remains active. If any of the membership conditions associated with the activated role fails, the role is deactivated. In some sense, the OASIS membership conditions are similar to our mid-conditions that must hold during operation execution.

RDL is not as generic and expressive as our approach and not as well suited to representing complex access control policies and those that include mandatory access control.

Policies, representable in Policy Maker and RDL, are restricted to the set of policies which do not produce side effects, resulting in change of the system state.

Ponder [4] is an object-oriented policy specification language that is suted for role-based access control policies, as well as general-purpose management policies. Ponder is targeted for different types of policies, including obligations, authorizations, delegation and filtering policies, and grouping these policies into aggregate structures. The obligation policies, for example, specify what actions (e.g., notification or logging) are carried out when specific events occur within the system. To some extent, the request-result and post-conditions in our framework serve a similar purpose. However, there are several significant differences between Ponder's and our approaches. First, in our framework all security requirements are expressed in a single policy structure, whereas in the Ponder approach authorization and obligation policies can be specified independently. These can lead to conflicts between the two policy types. Second, the policy in our framework is enforced by the same access control mechanism. The three-phase policy enforcement model allows for parts of policy (particular conditions) to be enforced at different times. In contrast, the Ponder uses a separate enforcement mechanism for each policy type.

Finally, the Ponder obligation policies are triggered by system events whereas in our framework the actions are triggered by other conditions in the same policy, such as threshold or system threat level.

Minsky and Ungureanu [11], [12] define the policy in terms of messages that only a restricted set of agents is permitted to exchange. Furthermore, the message exchange is controlled by a set of rules that is included in the policy. The policy enforcement mechanism is based on a set of trusted agents that interpret the rules and enforce them by regulating the message exchanges and the effect that the messages have on the control state (attributes and permissions) of the participating agents.

The ability to communicate and change the state resembles our concept of the read and write conditions. Our approach is different in that the "state" has a wider meaning. It includes all security-relevant information about real world which is representable in a computer system, e.g., bank account balance, temperature and user identity. Another difference is that the reading and writing of the state is based on the ordered synchronous evaluation of the conditions, rather than controlled message exchange.

Jajodia et al. [9] have proposed a logical language for the specification of authorizations. The concerns addressed in this work are orthogonal to the ones in this paper. In particular, they focus on modeling conflict resolution, integrity constraint checking and derivation rules (that derive implicit authorizations from explicit ones), while our work focuses on the representation and enforcement of authorization policies enhanced with detection and management of security violations.

Summary of the research of audit-based intrusion and misuse detection is given by Lunt [10]. Sandhu and Samarati [17] discuss authentication, access control and intrusion detection technologies and suggest that combination of the techniques is necessary in order to build a secure system.

5 Conclusions and Future Work

Traditional authorization mechanisms check whether a user is acting within prescribed parameters and will not detect abuse of privileges. Advanced policies can conditionally generate audit records and in limited ways can react to state generated by intrusion detection engines based on observation of the audit records. Such policies can also adapt the level of detail of the audit records generated until an intrusion detection engine notices that something is amiss, though not necessarily what it is. Such policies can also adapt the applied authentication policies to require more information from a user when suspicious activity has been detected.

In this paper we presented an authorization framework that enables the specification and enforcement of advanced authorization policies.

The GAA-API implementation is available at
http://www.isi.edu/gost/info/gaaapi/source.
For further details about the authorization model see [16].
For more information about the GAA-API see [15].

The GAA-API has been integrated with several applications, including ssh and Globus Security Infrastructure [6]. Currently we are integrating the GAA-API with FreeS/WAN IPsec.

There are some aspects of distributed policy evaluation and enforcement that do not fit well within the framework. In the current framework we assume that conditions are evaluated consecutively and that authorization requests do not overlap. These two assumptions enable us to concentrate on a single condition evaluation at a time and, therefore, avoid the problem of coordination of multiple condition evaluation processes.

This results in inefficient policy evaluation process and leads to systems that cannot scale to large numbers of objects. Our current approach may be appropriate for some client-server applications, where the server is an autonomous agent, in complete charge of its resources. The server maintains the security policy and is responsible for the policy evaluation. Some distribution of the policy evaluation process can be achieved through the condition evaluation function implemented as, for example, an RPC call that is performed synchronously. However, this approach is not suitable for truly distributed architectures where a set of servers implement the policy and the policy evaluation processing can be distributed over several servers. Each server is responsible for enforcing of a part of the whole access control policy.

The future directions for this research include exploring extensions to the framework that could encompass these issues.

6 Appendix

We use the Backus-Naur Form to denote the elements of our policy language. Curly brackets, {}, surround items that can repeat zero or more times. A vertical line, |, separates alternatives. Items inside double quotes are the terminal symbols. An EACL is specified according to the following format:

```
eacl ::= {eacl_entry}
eacl_entry ::= pos_access_right conditions |
neg_access_right conditions
pos_access_right ::= "pos_access_right"
def_auth value
neg_access_right ::= "neg_access_right"
def_auth value
conditions ::= pre_conds mid_conds rr_conds
post_conds
pre_conds ::= {condition}
mid_conds ::= {condition}
rr_conds ::= {condition}
post_conds ::= {condition}
condition ::= cond_type def_auth value
cond_type ::= alphanumeric_string
def_auth ::= alphanumeric_string
value ::= alphanumeric_string
```

`cond_type` defines the type of condition, e.g., access identity or time.

`def_auth` indicates the authority responsible for defining the value within the `cond_type`, e.g., Kerberos.

`value` is the value of condition. Its semantics is determined by the `cond_type` field. The name space for the value is defined by the `def_auth` field.

It should be pointed out that the EACL language description presented here is not complete. Our current framework supports flexible policy composition model. The discussion of this issue is beyond the scope of this paper.

Next we present an example of an EACL that governs access to a host.

Entry 1 specifies that Tom can not login to the host.

Entries 2 and 3 mean that logins from the specified IP address range are permitted, using either X509 or Kerberos for authentication if the number of previous login attempts during the day does not exceed 3. If the request fails, the number of the failed logins for the user should be updated. The connection duration time must not exceed 8 hours.

Entry 4 means that anyone, without authentication, can check the status of the host if he connects from the specified IP address range.

Entry 5 specifies that host shut downs are permitted, using Kerberos for authentication. If the request succeeds, the user ID must be logged. If the operation fails, the sysadmin must be notified by e-mail.

```
    # EACL entry 1
neg_access_right test host_login

pre_cond_access_id KerberosV.5 tom@ORGB.EDU
    # EACL entry 2
pos_access_right test host_login

pre_cond_location IPsec 10.1.1.0-10.1.200.255
pre_cond_access_id X509
"/C=US/O=Trusted/OU=orgb.edu/CN=partnerB"
pre_cond_threshold local ≤3failures/day/failed_log/
rr_cond_update_log local on:failure/failed_log/info:userID
mid_cond_duration local ≤8hrs
    # EACL entry 3
pos_access_right test host_login

pre_cond_location IPsec 10.1.1.0-10.1.200.255
pre_cond_access_id KerberosV.5 partnerb@ORGB.EDU
pre_cond_threshold local ≤3failures/day/failed_log/
rr_cond_update_log local on:failure/failed_log/info:userID
mid_cond_duration local ≤8hrs
    # EACL entry 4
pos_access_right test host_check_status

pre_cond_location IPsec 10.1.1.0-10.1.200.255
    # EACL entry 5
pos_access_right test host_shut_down

pre_cond_access_id KerberosV.5 trusted@ORGA.EDU
rr_cond_audit local on:success/info:userID
post_cond_notify local email/to:sysadmin/on:failure
```

7 Acknowledgement of Sponsorship

Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, and the Xerox Corporation under the following agreements: (1) F30602-00-2-0595, Dynamic Policy Evaluation of Containing Network Attacks Project (DEFCN); (2) DABT63-94-C-0034, Security Infrastructure for Large Distributed Systems Project (SILDS); (3) J-FBI-95-204, Global Operating System Technologies Project (GOST); (4) DE-FC03-99ER25397, Diplomat project; and (5) HE1254-97, XAUTH Project.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

8 Disclaimer

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, U.S. Department of Energy, the U.S. Government or the Xerox Corporation.

References

[1] M. Blaze, J. Feigenbaum and J. Lacy.
Decentralized Trust Management.
Proceedings IEEE Symposium on Security and Privacy, IEEE Computer Press, Los Angeles, pages 164-173, 1996.

[2] M. Blaze, J. Feigenbaum and M. Strauss.
Compliance Checking in the Policy Maker Trust Management System. *In Proceedings of the Financial Cryptography '98, Lecture Notes in Computer Science*, volume 1465, pages 254-274.

[3] M. Carney and B. Loe.
A Comparison of Methods for Implementing Adaptive Security Policies. *In Proceedings of the 7th USENIX Security Symposium*, pages 1-14, January, 1998.

[4] N.Damianou, N. Dulay, E. Lupu and M. Sloman.
The Ponder Policy Specification Language. *In Proceedings of the Workshop on Policies for Distributed Systems and Networks*, Springer-Verlag LNCS 1995, pages 18-39, Bristol, UK, January, 2001.

[5] I. Foster and C. Kesselman, editors.
The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, 1999.

[6] I. Foster and C. Kesselman.
Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, Summer 1997.

[7] R. J. Hayton, J. M. Bacon and K. Moody.
OASIS: Access Control in an Open, Distributed Environment.
Proceedings of the IEEE Symposium on Security and Privacy, pages 3-14, Oakland, CA, May 1998.

[8] Y. Huang and M. Shan.
Policies in a Resource Manager of Workflow Systems: Modeling, Enforcement and Management. *Hewlett Packard Lab Software Technology Laboratory Technical Report HPL-98-156*, September, 1998.

[9] S. Jajodia, P. Samarati and V.S. Subrahmanian.
A logical Language for Expressing Authorizations.
Proceedings of the 1997 IEEE Symposium on Security and Privacy, 1997.

[10] T. F. Lunt.
A Survey of Intrusion Detection Techniques, *Computers and Security*, volume 12, pages 405-418, June 1993.

[11] N. Minsky and V. Ungureanu.
Unified Support for Heterogeneous Security Policies in Distributed Systems. *In 7th USENIX Security Symposium,* San Antonio, Texas, January 1998.

[12] N. Minsky and V. Ungureanu.
Law-Governed Interaction: A Coordination & Control Mechanism for Heterogeneous Distributed Systems. *In ACM Transactions on Software Engineering and Methodology (TOSEM),* Vol 9, No 3, pages 273-305, July 2000.

[13] B.C. Neuman.
Proxy-based authorization and accounting for distributed systems. *In Proceedings of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, May 1993.

[14] B.C. Neuman and T. Ts'o.
Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, pages 33-38, September 1994.

[15] T. V. Ryutov and B. C. Neuman.
Representation and Evaluation of Security policies for Distributed system Services. *In Proceedings of the DARPA Information Survivability Conference and Exposition*, January 2000. Hilton Head, South Carolina.

[16] T. V. Ryutov and B. C. Neuman.
The Set and Function Approach to Modeling Authorization in Distributed Systems. *In Proceedings of the Workshop on Mathematical Methods and Models and Architecture for Computer Networks Security*, May 2001, St. Petersburg Russia.

[17] R. Sandhu and P. Samarati.
Authentication, Access Control, and Intrusion Detection.
The Computer Science and Engineering Handbook, pages 1929-1948, CRC Press, 1997.

Appendix E:

The Set and Function Approach to Modeling Authorization in Distributed Systems

Tatyana Ryutov and Clifford Neuman

Information Sciences Institute University of Southern California
4676 Admiralty Way suite 1001
Marina del Rey, CA 90292
{tryutov, bcn}@isi.edu
(310)822-1511 (voice) (310)823-6714 (fax)

Abstract. We present a new model that provides clear and precise semantics for authorization. The semantics is independent from underling security mechanisms and is separate from implementation. The model is capable of representing existing access control mechanisms. Our approach is based on set and function formalism. We focus our attention on identifying issues and use our model as a general basis to investigate the issues.

1 Introduction

The Internet has rapidly evolved to a platform that supports business and services such as e-commerce, electronic publishing, and health care. Security compromises now have real world consequences, resulting in release of sensitive or protected information and monetary loss. Attacks on medically critical computing capabilities might even result in loss of human life. The ability to define and enforce fine-grained security policies for systems and services is important in such systems. The ability to understand such security policies is critical if they are to be correctly written or implemented. Unfortunately, as the complexity of the systems grow, these polices are becoming harder to correctly define and more difficult to enforce.

To cope with the growing complexity of policy specification it is useful to design a conceptual model that gives a structured way to think about policies. A model enables one to better understand the domain of study, visualize the main elements and their behavior at some chosen level of detail and use a short hand notation for precise description and decreased ambiguity. Furthermore, the conceptual integrity of a system derives from a coherent high-level view of the system organization and functionality. Thus, one of the main objectives of this work is to construct a conceptual model for policy representation and evaluation. For doing so, we use a methodology based on concepts of sets and functions.

In our paper we are only interested in the class of authorization policies versus a wider range of policies, such as distributed system management policies. The goal of authorization polices is to govern access to objects. Supporting such policies takes the form of monitoring and restricting the user activity within the distributed system (access control), making authorization decisions (authorization) and performing necessary actions to modify the behavior of the system (policy enforcement).

An authorization policy specifies conditions, which must be satisfied before, during or after the access right is exercised. For example, it may be desirable to enforce the following policy: “A process can be run on the host A if the request originates from a domain B and the process does not use more then 20% of the CPU time. An audit record about the started process must be generated”.

This policy specifies several conditions:

1. *location of the requester*
   This condition must be satisfied before the access right ”process run” is granted.
2. *system load*
   This condition must hold while the process is running.
3. *audit record generation*
   This condition must be met after the process is started.

Our model captures this intuitive notion of authorization policy and provides a formalism for the policy representation and evaluation.

There has been extensive research in authorization and a number of formal models have been developed.

Some of these contributions focus on addressing authorization requirements for specific policy domains, e.g., database systems [3], collaborative environment [17] or separation of duty [2]. Others are concerned with a particular access control mechanism, such as an ACL [1].

What is still missing, is a unified view of authorization in a distributed, multi-policy environment. Such a environment is composed of connected independent computer systems managed by separate administrative authorities. In a multi-policy environment the policy integration should incorporate diverse authorization models, which can coexist in a distributed system. Administrators of each domain might express security policies by means of different formalism.

Generalizing the way that applications define their authorization requirements provides the means for integration of local and distributed security policies and translation of security policies across multiple authorization models.

Our paper describes an authorization model designed to meet these needs. In particular, our model allows us to represent existing access control models (e.g., ACL and capability) in a uniform and consistent manner.

The model simplifies the specification of complex authorization policies and provides a generic policy evaluation environment. Furthermore, the model provides a general basis for identifying and resolving issues, not well-understood before, such as side effects of the policy evaluation on the system state and related policies.

By separating generic from domain specific elements, we ensure that the model is extensible to arbitrary (authorization policy) domains.

We keep our model simple and practical to serve as an aid to implementation. We have found that the model suggested ideas for implementations, for example that condition implementation should be based on three phases.

Our final goal is to implement a subset of our conceptual model and provide a programmable framework for different kinds of polices. The framework maps real-world policy entities such as users, resources, and organizational policies, to the representation of these entities in the programming environment. The discussion of the initial implementation can be found in [14].

2 Related work

In this section we review prior research in representation and evaluation of authorization. Formal semantics for policy representation and evaluation has been used by other researches, in particular Woo and Lam [15].

Their work addresses general concerns as ours, in particular, positive and negative authorizations and providing computable semantics. In our model, authorization is given a precise semantics independent of underlying policy requirements. This distinguishes our work from [15] where a formal notion of an authorization policy has different semantics for each set of authorization requirements.

The Policy Maker system described in the papers by Blaze, et al. [4], [5] focuses on construction of a practical algorithm for determining trust decisions. Policies and credentials encode a set of trust relationships among the issuing sources.

In Policy Maker's terminology, "proof of compliance question" asks if the request q, supported by a set of credentials complies with a policy p. This is equivalent to the authorization question that we consider in our work: "is request q authorized by the policy p (in our model credentials are contained in the request)". Their approach, however, is different from ours.

In our approach, the information passed to the authorization engine with the authorization request is used to evaluate conditions in the relevant policy statements. Each condition is evaluated just one time. The order of condition evaluation is important.

In Policy Maker, the credentials and policy (called assertions) are used collectively to compute a proof of compliance. The assertions can be run in arbitrary order (and possibly many times) and produce intermediate results, that then can be fed into other assertions. Policies, representable in the Policy Maker, are restricted to the set of policies which do not produce side-effects, resulting in change of the system state. The Policy Maker can be integrated in our model as a component for evaluation of the **trust constraints** conditions.

Detailed formal language specification based on set and function formalism is given in the paper by Sandhu [2] for specific constraints of separation of duty in role based environment. The language semantics is defined by a restricted form of the first order logic. The formal language provides a useful model to study properties of conflict of interests, in particular separation of duty.

The paper by Abadi, et al. [1] presents a logical language for access control lists. They study the notions of delegation, roles and groups using their logical language and rules for making access control decisions.

The exploratory work by Moffet and Sloman [11] is aimed to understanding policy semantics. The two aspects of a policy are considered: motivation and actual ability to carry out actions.

3 Basic Conceptual Model

The conceptual model presents the high level organizing principles of the authorization model and defines the strategy chosen to realize the model.

3.1 Policy Elements

In this section we explore the notion of a policy and abstract it into a conceptual model. This section prepares us for going to the more detailed specification given in the next section. We start the design of the conceptual model with specification of the components that are to be modeled. At a conceptual level a policy is a compound entity, which regulates access to objects.

The notion of an object is central to the policy definition. An object is a target of requests and it has to be protected. An object can be a physical resource such as a host or a communication channel, as well as an abstract, higher level entity, e.g., a bank account.

An access right is a particular type of access to a protected object, e.g., read or write. The notion of a negative access right is useful to specify many practical policies. Sometimes it is easier to allow access to all and explicitly disallow access for those who should not have access.

A condition describes the context in which each access right is granted. A condition must be satisfied in order to allow an operation to be performed on a target object[1]. Here are several of the more useful conditions [12].

- **access identity**
  Specifies an authenticated access identity (subject) on whose behalf request to access an object has been issued.
- **time**
  Time periods for which access is granted.
- **location**
  Location of the principal. Authorization is granted to the principals residing on specific hosts, domains, or networks.
- **payment**
  Specifies a currency and an amount that must be paid prior to accessing an object.

[1] However, if the access right is negative, the access is denied if all conditions are met.

- **quota**
  Specifies a currency and a limit. It limits the quantity of a resource that can be consumed or obtained.
- **audit**
  Enables automatic generation of an application level audit data in response to access requests.
- **notification**
  Enables automatic generation of notification messages in response to access requests. Specifies the notification method and a receiver.
- **trust constraints**
  Specifies restrictions placed on security credentials. Allows one to validate the legitimacy of the received certificate chain and the authenticity of the specified keys.
- **attributes of subjects**
  Defines a set of attributes that must be possessed by subjects in order to get access to the object, e.g., user age.

Traditional security thinking has been oriented toward authentication as a prerequisite for authorization. Usually authorization applies after authenticated requester identity has been established.

In our model policies are treated as the first class citizens. Authentication, audit and accounting mechanisms are activated by explicit policy requirements, expressed through conditions. If a policy does not require authenticated user identity, authentication steps can be ignored or deferred until the policy explicitly requests it. An example of a policy, which is not concerned with the identity is "anyone can read file A if \$10 is paid".

Note that in the implementation, some of these conditions might have side effects. For example, evaluation of **payment** and **quota** conditions reduces a balance somewhere. Evaluation of **notification** condition results in sending a message, which is useful in audit.

Unfortunately, side effects might complicate the model. Ignoring the side effects might cause problems when the side effects create a feedback loop, for example, when an audit record triggers a network threat detection which affects the evaluation of subsequent policies, or where payment affects quotas which affects the ability to perform other operations (once one runs out of money).

Balancing the complexity this adds with the simplicity of the model is still an open issue, which requires further investigation. Initial ideas on handling the side effects are given in Section 4.2.

3.2 Basic Definitions and Assumptions

We present our conceptual model based on set and function formalism, algebra of sets and first order logic. The conceptual model specification is guided by conventional authorization notions and expected authorization requests.

An elementary policy statement consists of an object component, a positive or negative access right component and zero or more condition components. Thus, to represent the components, we define sets of elements called objects O, positive rights R, negative rights $\overline{R}$ and conditions C. All existing policy statements are contained in the set P. In addition, we define a set of authorization requests Q.

All the sets, except for C[2] , are finite dynamic and unordered. The dynamic property means that sets are not fixed, new elements can be added and existing elements can be deleted. The finite property assumption requires that at any particular time, the sets are finite. Negation is applied only to the elements of the set R to model negative rights. We do not define negative conditions. The empty set is denoted by $\emptyset$.

O is finite dynamic non-empty unordered set of object elements:

$$O = \{o_1, o_2, \ldots, o_n\} \ . \tag{1}$$

R is finite dynamic non-empty unordered set of access right elements:

$$R = \{r_1, r_2, \ldots, r_n\} \ . \tag{2}$$

$\overline{R}$ is finite dynamic non-empty unordered set of negative access right elements. Set $\overline{R}$ is constructed from the set R by applying negation to each element of the set R.

$$\overline{R} = \{\neg r_1, \neg r_2, \ldots, \neg r_n\} \ . \tag{3}$$

Note that $R \bigcap \overline{R} = \emptyset$.

C is dynamic unordered set of condition elements with a special condition element c^*, which represents an empty condition:

$$C = \{c^*, c_1, c_2, \ldots, c_n\} \ . \tag{4}$$

P is finite dynamic unordered set of compound policy elements:

$$P = \{p_1, p_2, \ldots, p_n\} \ . \tag{5}$$

Each element p of the set P represents a set of three elements:

$$p = \{o, r, c\} \, , \ o \in O, \quad r \in R \cup \overline{R}, \ c \in C \ . \tag{6}$$

Note that a condition element can be c^*. When $c = c^*$ the rights are granted or denied unconditionally. An example of a practical policy with an empty condition is: “file A can be read by anyone”.

Q is finite dynamic partially ordered set[3] of compound authorization request elements:

$$Q = \{q_1, q_2, \ldots, q_n\} \ . \tag{7}$$

Each element q of the set Q represents a set of three elements:

$$q = \{o, r, c\}, \ o \in O, \ r \in R, \ c \in C \ . \tag{8}$$

[2] The conditions can be represented by different entities, including numbers (see Section 4.2), so we can not state finiteness property.

[3] The reasoning behind the requirement of the partial ordering of the set Q is discussed in Section 4.2.

The elements correspond to the target object (o), requested access right (r) and a condition constant (c). The condition constant c represents information which is matched to the requirements specified in the condition of the relevant policy statement. In practice, this information can be represented by a set of credentials, e.g., authenticated user identity. For example, a policy statement "Anyone can read file A from 8am till 6pm" specifies a **time** condition. The request "read (r) file A (o) at 5pm (c)" specifies current time and is matched to the **time** condition in the policy statement.

To make our model practical, special provisions should be made for dealing with the following situations:

- incomplete data, not known at the authorization time. During network fragmentation some data may be inaccessible.
- policy requires a certain event to happen in the future. Statements about the future do not have truth values until the event described takes place.
- the function used to evaluate conditions does not terminate for the arguments supplied. Incorrect implementation, bad parameters.

In order to properly deal with these situations we will adopt a three-valued logic [9], [13].

Three-valued logic is classical boolean (true/false) logic extended with a third truth value - undefined.

We define an auxiliary set B, consisting of the three constants: true, represented by T, false, represented by F and U, meaning uncertainty.

$$B = \{T, F, U\} \ . \tag{9}$$

Table 1 shows the truth tables, when at least one argument is equal to U.

P	Q	P & Q	P $\vee$ Q
T	U	U	T
U	T	U	T
F	U	F	U
U	F	F	U
U	U	U	U

Table 1.

In addition, $\neg U = U$. Next we define functions to express an authorization process.

The *by_object* function takes a set of policy elements P and request q, which contains particular object $\widehat{o}$ as an argument and returns a subset $P' \subseteq P$ where this object appears.

$$P' = by\_object(P, q),$$

$$\widehat{o} \in O,\ \widehat{o} \in q,\ q = \{\widehat{o}, r, c\},\ \ q \in Q,\ P' \subseteq P : \forall p' \in P' : p' = \{\hat{o}, r, c\} \ . \tag{10}$$

The *by_right* function takes a set of policy elements P and request q, which contains particular access right $\widehat{r}$ as an argument and returns a subset $P' \subseteq P$ where this right appears.

$$P' = by\_right(P, \widehat{r}),\ \widehat{r} \in R,$$

$$P' \subseteq P : \forall p' \in P' : p' = \{o, \widehat{r}, c\}\ or\ p' = \{o, \neg\widehat{r}, c\}\ . \tag{11}$$

The *eval_cond* is a condition evaluation function.

$$b = eval\_cond(\widehat{c}, \widetilde{c}),\ \widehat{c} \in C,\ \widetilde{c} \in C,\ b \in B\ . \tag{12}$$

The function M defines positive or negative modality of the policy element. If the access right, contained in the policy element is positive or negative, the modality is positive or negative, respectively.

$$M(p_i, q) = \begin{cases} eval\_cond(\widehat{c}, \widetilde{c}), & \widehat{r} \in R \\ \neg eval\_cond(\widehat{c}, \widetilde{c}), & \widehat{r} \in \overline{R}, \end{cases}$$

$$\widehat{c} \in p_i,\ \widetilde{c} \in q,\ \widehat{r} \in p_i,\ p_i \in P',\ q \in Q\ . \tag{13}$$

The M function has to be applied to all elements $P' \subseteq P$. The evaluated modality of each policy element will be taken with or without the negation $\neg$ according to its right. After all the modalities are evaluated, we will take their disjunction. These operations are performed by the *eval_conditions* function.

$$b = eval\_conditiods(P', q) = M(p_1, q) \bigvee M(p_2, q) \bigvee ... \bigvee M(p_n, q),$$

$$p_i \in P',\ i = \overline{1, n},\ n\ is\ the\ cardinality\ of\ P',$$

$$P' \subseteq P,\ q \in Q,\ b \in B\ . \tag{14}$$

The resulting value b obeys to the $\bigvee$ operation for three-valued logic. That is, *eval_conditions* returns T if at least one modality gave the result T, F if all results were F, and U otherwise (i.e., at least one result was U, possible some F but none T).

The *authorization* is a composite function:

$$b\ =\ authorization(P, q)\ =$$

$$=\ eval\_conditions(P''', q)\ \circ\ by\_right(P', q)\ \circ\ by\_object(P, q)\ =$$

$$=\ eval\_conditions(P''', q)\ \circ\ by\_object(P', q)\ \circ\ by\_right(P, q)\ . \tag{15}$$

The *authorization* function takes the set of policies P and an authorization request q as arguments. It returns F, T or U meaning authorized, not authorized or uncertain. Three-valued logic at the conceptual level has to be mapped to the two-valued logic at the implementation level. In the end, the access must be either granted or denied.

3.3 Time Dependency

Time dependency appears in our conceptual model implicitly. At each instant only the set of policies which exists at authorization time is considered. All future or past policies are irrelevant. Note that this does not mean that the current policy does not depend on the past or future events. Some policies must take into account the system execution history or the fact that particular event must have happened for some operation to take place. An example of practical policy taking into account occurrence of some event is "If one reads file A, then one can not send" [16]. Some policies may need to know precise time of the event occurrence, for example for audit purposes. This may require a time-stamping of certain occurrences and keeping record of them.

3.4 Changes in the Set Membership

Exercising access rights can result in creating new objects and defining new policies. In the conceptual schema this is represented as adding an element to the corresponding set. As we discussed in the previous section, changes in membership of the sets R and $\overline{R}$ depend entirely on the set O.

The deletion of an element from the sets O, R or $\overline{R}$ entails deletion of each element from P in which the deleted element appears. To simplify our model we require that rights can be applied only to the elements of set O. If we allow rights to be applied to the elements of P, we will have to consider a policy management model.

3.5 Policy Representation Issues

We do not allow use of the disjunction in representation of elements of the set P. The disjunctive form policies such as "Tom or Joe can read file A", "Tom can read either file A or B" and "Tom can either read or write file A" is modeled by using separate policy statements.

$$O = \{A, B\}, R = \{read, write\}, C = \{c^*, Tom, Joe\},$$

$$P = \{\{A, read, Tom\}, \{A, read, Joe\}, \{B, read, Tom\}, \{A, write, Tom\}\} .$$

However, disjunction of policy elements can be used in practice for optimization reasons. For example, in the implementation of an ACL we can combine several access rights which correspond to a particular access identity condition.

Let us consider the exclusive OR policy representation: "Tom can read files A or B, but not both". This policy is a variant of the Chinese wall policy [6], required in the operation of many financial services. The policy guards against the conflict of interest. A consultant can freely chose a company in order to offer an advice. However, once the company has been chosen, the consultant is mandatory denied access to the information about all other companies. This policy can be implemented using an additional condition, let us call it *trigger_history*. This condition activates the history of execution.

$$P = \{\{A, read, Tom, trigger\_history\}, \{B, read, Tom, trigger\_history\}\} \ .$$

If Tom decides to read file A first, the history is checked, and since initially it is empty, the right is granted and the information about it is stored. If he tries to read file B after that, the request will be denied. A history information is maintained by the system. The history can be centralized or distributed. An example of implementation of the condition is briefly described in [18]. More detailed discussion of implementation of the history-dependent access control policies is given in [10].

In conventional access control models, a subject has been a separate notion. A subject is an entity on whose behalf a request to access an object has been issued. Traditionally, policy conceptualization is based on three basic entity types: objects, access rights and subjects. Some of the possible logical groupings of these entities, such as ACL and capability, have become practical implementations of the Lampson matrix [8].

In the ACL based systems, policies are grouped by objects. A typical ACL is associated with an object (or a group of objects) to be protected and enumerates the list of authorized subjects and their rights to access the object.

In the capability-based systems, policies are grouped by subjects. A capability lists sets of objects accessible by the subject along with the types of access rights.

These logical grouping can be represented in our model.

ACL An ACL consists of a set of ACL entries. An ACL entry is analogous to a policy element p, where all conditions are **access identity**.

Consider a policy: “Tom and Bob can read and write file A”. We can translate this policy into our policy model as:
“Tom (condition c_1) and Bob (condition c_2) can read (positive right r_1) and write (positive right r_2) file A (object o_1) “. We need four policy elements to represent this policy:

$$\begin{aligned} p_1 &= \{o_1, r_1, c_1\}, \\ p_2 &= \{o_1, r_2, c_1\}, \\ p_3 &= \{o_1, r_1, c_2\}, \\ p_4 &= \{o_1, r_2, c_2\} \ . \end{aligned}$$

This way of specification and storage of the policy is tedious and inefficient.

To represent an ACL, we adopt three modifications to the representation of a policy element p specified in(6):

1. An ACL is associated with each object, so the object is implicit and is omitted from the policy elements.
2. Conditions are listed first, then access rights. This order is closer to the traditional ACL specification.
3. We allow disjunction of either positive or negative access rights.

Now we need only two ACL entries to represent the policy:

$$p_1 = \{c_1, r_1 \vee r_2\},$$
$$p_2 = \{c_2, r_1 \vee r_2\} .$$

Furthermore, if we allow conditions to be aggregated into a single entry when the same set of access rights applies to all of them, we need only one policy statement to represent the policy: $p_1 = \{c_1 \vee c_2, r_1 \vee r_2\}$.

by_object function returns all policy statements associated with the given object. The returned set of policies P' conceptually represents an ACL associated with the object $\widehat{o}$.

Capability To demonstrate how capabilities can be represented, we define function *by_condition*, which takes the set of policies P and particular condition $\widehat{c}$ as arguments and returns a subset P', where this condition appears. Intuitively, this function returns all policy statements associated with the given condition.

$$P' = by\_condition(P, \widehat{c}),\ \widehat{c} \in C,\ P' \subseteq P : \forall p' \in P' : p' = \{o, r, \widehat{c}\} .$$

Note that if the condition constant $\widehat{c}$ specifies particular access identity (subject), then the returned set of policies P' conceptually represents a capability possessed by the subject identified by the condition $\widehat{c}$. Next the set P' can be passed to the *authorization* function along with an authorization request for further evaluation.

Representation of a capability is quite similar to that of an ACL. A capability is associated with each subject, so the subject is implicit and is omitted from the policy element. Thus, each policy statement contains only elements, which represent objects and access rights.

More detailed discussion of the implementation of ACL and capability can be found in [14].

4 Extended Conceptual Model

The extended conceptual model expands upon basic conceptual model entities and interactions. The notion of a policy hierarchy is introduced. The design work at this level addresses condition side-effects issues.

4.1 Refinements

In this section we describe further refinements of our basic entities. A policy statement may specify several conditions of different types, for example: “Tom can read file A only between 9am and 6pm”. This policy defines two conditions: **access identity** and **time**. In (6) we have considered only one condition in the policy statement. All existing conditions were aggregated into one set (4). Now we extend the notion of a condition to be distinguished not only by an identifier but also by a type. Each condition element has just one type. We assume that at each instant S condition types exist. We represent these different condition types by S disjoint sets:

$$C = \bigcup^{k=\overline{1,S}} C^k,\ C^i \bigcap^{i,\,j=\overline{1,S}\; i\neq j} C^j = \emptyset\ . \tag{16}$$

Now we define a totally ordered[4] set $\widetilde{C}$. Each element of this set is constructed from one element of the S disjunctive sets. Intuitively this means that each element of $\widetilde{C}$ consists of S condition elements of different types, some of the elements can be c^*.

$$\widetilde{C}\ =\ \{\widetilde{c}^1, \widetilde{c}^2, ..., \widetilde{c}^S\},\ \widetilde{c}^i \in C^i,\ i = \overline{1,S}\ . \tag{17}$$

We define S condition evaluation functions for each condition type. In our policy example we define two function for checking access identity and current time.

$$b\ =\ eval_i(\widetilde{c}^i, \widetilde{c}^i),\ \widetilde{c}^i \in C^i,\ \widetilde{c}^i \in C^i,\ b \in B,\ eval_i(c^*, c^*) = T,\ i = \overline{1,S}\ . \tag{18}$$

From (4.2) and (15) we observe that if at least one of the policy statements evaluates to T, the authorization will be granted. This behavior may not be always desirable. For example, we would want a policy assigned by the system administrator to take precedence over the one assigned by an individual user. This requires the means of specifying a hierarchical relationship among policy statements.

The hierarchy of policies is modeled by assigning priorities. We do not attempt to give a full theoretical development of the method of assigning priorities here. The essential requirements is that one should be able to decompose the whole policy into totally ordered policy statements. To express policy priorities, we define set W. W is a finite totally ordered set of elements that can be compared (e.g., integers).

$$W = \{w_1, w_2, \ldots, w_n\}\ ,\ \ w_i < w_j,\ i,j = \overline{1,L},\ i < j,\ L\ is\ the\ cardinality\ of\ W\ .$$

We redefine element q, given in (8) in the following way:

$$q = \{o, r, \widetilde{c}^1, \widetilde{c}^2, ..., \widetilde{c}^S\},\ o \in O,\ r \in R,\ \widetilde{c}^i \in C^i,\ i = \overline{1,S},\ q \in Q\ . \tag{19}$$

We extend (6) in two ways: 1) each element p has an additional component w, which denotes priority of this element. 2) condition component is represented by a set of $\subseteq$ condition constants of different types.

$$p = \left\{o, r, \widetilde{C}', w\right\},\ o \in O,\ r \in R \bigcup \overline{R},\ \widetilde{C}' \subseteq \widetilde{C},\ w \in W,\ p \in P\ . \tag{20}$$

Figure 1 illustrates representation of a policy element p.

[4] The reasoning behind the requirement of the total ordering of the set $\widetilde{C}$ is discussed in Section 4.2.

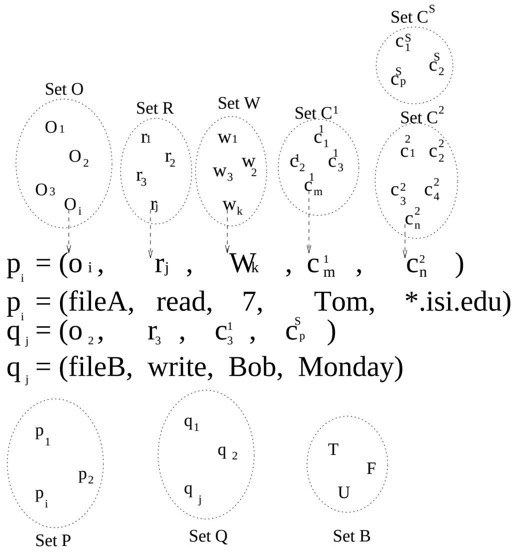

Figure 1.

The *by_priority* function takes a set of policies P as an argument and returns a subset P' with the maximum priority. The ordering in the set W determines the policy statement which is enforced if several policy statements are simultaneously satisfied. Note that if the set P' contains more then one element, the elements have equal priorities. In this case, if any of the policy statements is satisfied, authorization is granted.

$$P' = by\_priority(P),$$

$$P' \subseteq P : \forall p' \in P',\ p' = \{o, r, \widetilde{C}, \widehat{w}\},\ \widehat{w} = \max(\forall w : p = \{o, r, \widetilde{C}, w\} \in P') \ . \tag{21}$$

We redefine *eval_cond* function given in (12) in the following way:

$$eval\_cond \ = \ eval_1(\widehat{c}^1, \widetilde{c}^1)\&eval_2(\widehat{c}^2, \widetilde{c}^2)\&...\&eval_S(\widehat{c}^s, \widetilde{c}^s),$$

$$\widehat{c}^i \in C^i,\ \widetilde{c}^i \in C^i,\ i = \overline{1, S},$$

$$b = eval\_cond(p),\ p \in P,\ b \in B \ . \tag{22}$$

The *eval_cond* function is a short hand notation for representation of conjunction of the results, obtained by applying $eval_i$ to corresponding condition constants from the policy element p. All conditions must be met simultaneously in order to satisfy the authorization request.

The resulting value b obeys to the & operation for three-valued logic. That is, *eval_cond* returns T if all elements gave the result T, F if at least one result was F, and U otherwise (i.e. at least one result was U, possible some T but none F.

We redefine *authorization* function given in (15) in the following way:

$$b \;=\; authorization(P, q) \;=$$

$$= \; eval\_conditions(P''', q) \circ by\_priority(P'') \circ by\_right(P', q) \circ by\_object(P, q) \;=$$

$$= \; eval\_conditions(P''', q) \circ by\_priority(P'') \circ by\_object(P', q) \circ by\_right(P, q),$$

$$P''' \subseteq P'' \subseteq P' \subseteq P, \; q \in Q, \; b \in B \; . \tag{23}$$

Figure 2 illustrates the *authorization* function.

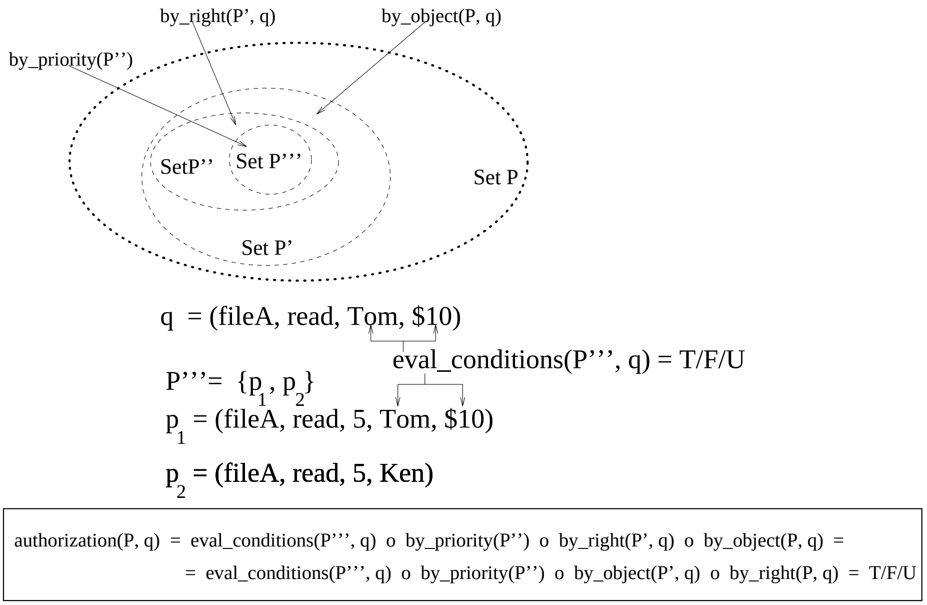
by_right(P', q)
by_object(P, q)
by_priority(P'')
SetP''
Set P'''
Set P
Set P'
q = (fileA, read, Tom, $10)
eval_conditions(P''', q) = T/F/U
P'''= {p1, p2}
p1 = (fileA, read, 5, Tom, $10)
p2 = (fileA, read, 5, Ken)
authorization(P, q) = eval_conditions(P''', q) o by_priority(P'') o by_right(P', q) o by_object(P, q) =
= eval_conditions(P''', q) o by_priority(P'') o by_object(P', q) o by_right(P, q) = T/F/U

Figure 2.

4.2 Discussion of Condition Side-Effects

The total order property of the set $\widetilde{C}$ defined in (16) requires that policy elements that differ only by the order of condition elements are considered to be distinct. This property is important to deal with possible side effects caused by the condition evaluation. Consider a policy "Tom can read file A only if notification is sent (**notification** condition) and system threat condition is low (*threat_level_low* condition)". Assume that current system threat level is low. Assume that the notification about Tom reading file A triggers high system threat level. There are two ways to represent the policy in our model:

$$p_1 = \{A, read, Tom, threat\_level\_low, notification\},$$
$$p_2 = \{A, read, Tom, notification, threat\_level\_low\} \; .$$

The evaluation of p_1 results in access grant, however evaluation of p_2 results in denial.

In this section we will discuss determining the correct order of the condition elements in the policy statement p defined in (20).

System State Representation To discuss side effects produced by evaluation of some conditions, we introduce time into our model explicitly. Time is discrete and is represented by a totally ordered set of natural numbers. Each number corresponds to a discrete time interval. A time interval is related to a condition evaluation process.

To simplify our presentation, we assume that dependent authorization requests do not overlap. The effects of the dependent requests are resolved by serialization, in which the requests are ordered by the cause-effect ordering.

Similarly, we assume that conditions are evaluated consecutively. These two assumptions enable us to concentrate on a single condition evaluation per each time interval and, therefore, avoid the problem of coordination of multiple condition evaluation processes.

Figure 3 illustrates our representation.

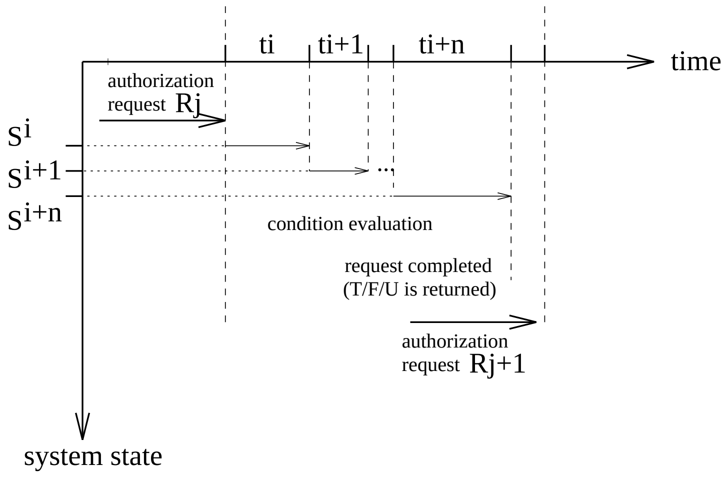

Figure 3.

A time interval begins when a condition evaluation starts and it ends when the condition evaluation is completed with the resulting $T/F/U$. This means that the duration of the time intervals can vary.

The general idea underlying our approach is that the system state can be formalized by a sequence of system states S^1, S^2, …, S^k. Each system state S^i is labeled by the time interval i.

By a system state we mean not only information describing a particular computer system such as system load, network bandwidth consumption, number of available processors, but also all information about the real world which is representable in a computer system, for example: bank account balance, temperature, user identity.

Here any system state S^i is all the information that has been deduced up to the time interval i. The information is represented by a set of system variables. The information is partial, since some system variables can be undefined at some time intervals.

At each time interval i there is a transition $S^i \rightarrow S^{i+1}$from the current system state S^i to the new system state S^{i+1}. Each transition is characterized by updating the values of some system variables. The variables can change not only as the result of condition evaluation but also because of other events, e.g., system load is altered. All side effects of condition evaluation are recorded in the corresponding system variables.

Classification of Conditions In this section we present a taxonomy of conditions. We say that a condition **writes system state**, if the condition evaluation function changes values of some system variables.

The fact that evaluation of condition c changes value of the system variable j is represented by the notation $c(S^i) \rightarrow S_j^{i+1}$.

We say that a condition **reads system state** if the condition evaluation function requires reading of particular system variables.

The fact that evaluation of condition c requires the value of the system variable j is represented by the notation $c(S_j^i) \rightarrow S^{i+1}$.

We say that a condition $\widehat{c}$ **depends on condition** $\widetilde{c}$, if condition $\widehat{c}$ requires reading of some system variables, which are written by the condition $\widetilde{c}$.

The fact that condition $\widehat{c}$ requires the value of the system variable j, which is written by the condition $\widetilde{c}$ is represented by the notation: $\widetilde{c}(S_j^i) \hookrightarrow \widehat{c}(S_j^{i+1})$.

Conditions are classified by the read/write system state property:

– **read conditions** read system state but do not write system state, for example **time**, **location** and **system load**.
– **write conditions** write system state and may read system state, for example, **payment**. Payment requires checking for the presence of required amount (read system variable k) and reducing the balance by the requested amount (write system variable k). This is represented as: $c(S_k^i) \rightarrow S_k^{i+1}$.

Note that **write conditions** must be evaluated before the **read conditions** that are dependent on them.

Designing the condition ordering algorithm that satisfies the ordering requirements falls into the realm of scheduling of processes with precedence constraints and is outside of the scope of this paper.

Condition Representation and Evaluation Read conditions such as **access identity** and **location** appearing in the authorization request, specify a set of constants which must be matched against a corresponding set of constants found in the policy elements. These conditions are represented by a set C^1. This set is constructed from a set of all condition constants passed in authorization requests q defined in (19), a set of all condition constants contained in policy elements p defined in (20) and a set of operations M. Condition evaluation function for this type of conditions returns T if applying operation m, $(m \in M)$ to the condition constants evaluates to T, otherwise it returns F.

For example, a set of operations M may contain $(\subseteq)$. If $m = \subseteq$, condition evaluation function returns T if $\widehat{c}^1 \subseteq \widetilde{c}^1, (\widehat{c}^1 \in C^1, \widetilde{c}^1 \in C^1)$, otherwise it returns F.

Some conditions, such as **system load**, can be represented numerically. These conditions are evaluated by comparing numbers (natural, integer or real). Therefore, we can define the set of operations as
$M = \{ =, \neq, <, >, \leq, \geq \}$.

Write conditions, such as **notification** and **audit** specify the name of a system variable, whose value must be changed, and the new value. Condition evaluation function for these conditions returns T if the updating of the system variable succeeded[5], and F otherwise.

Unfortunately not all conditions can be represented in this way. In practice, conditions can be application-specific and complex. The problem is how an informal specification of the condition can be transformed into a precise formal mathematical structure, within which we can actually prove things about the properties, such as computability and polynomial-time decidability.

5 Conclusions and Future Work

In this paper we presented a conceptual model for authorization in distributed systems. We introduced precise semantics for policy representation and evaluation. The semantics is defined independently from underling security mechanisms and is separate from implementation. The flexibility of the model makes it possible to represent existing access control mechanisms.

We believe that the model provides an effective way to understand and employ authorization policies in distributed systems.

We have begun to investigate the side-effects of the condition evaluation. Through the use of the side effects, in our current work we consider integrating intrusion and misuse detection systems with applications using our model.

We hope that this model will lead to other insights about authorization policies. We are looking for possible ways to restrict condition expressiveness to guarantee policy computability and polynomial-time decidability.

[5] Updating the system variable can fail due to various reasons, for example we might be unable to append audit information to the audit log because the disc space has been exceeded.

6 Acknowledgments

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-0595, Security Infrastructure for Large Distributed Systems (SILDS) Project under agreement number DABT63-94-C-0034 and by Xerox Corporation, XAUTH Project under agreement number HE1254-97. The U.S. Government is authorized to reproduce and distribute repreints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, the U.S. Government, or Xerox Corporation.

References

1. Abadi, M., Burrows, M., Lampson, B. and Plotkin, G.: A calculus for Access Control in Distributed Systems. *ACM Transactions on Programming Languages and Systems, Vol. 15, No 4* (September 1993) 706–734
2. Gail-Joon Ahn and Sandhu, R.: The RSL99 Language for Role-Based Separation of Duty Constraints. *ACM Workshop on Role-Based Access Control* (1999) 43–54
3. Bertino, E. and Jajodia, S.: Supporting Multiple Access Control Policies in Database Systems. *Proceedings of the 1996 IEEE Symposium on Security and Privacy* (1996)
4. Blaze, M., Feigenbaum, J. and Lacy, J.: Decentralized Trust Management. *Proceedings IEEE Symposium on Security and Privacy*, IEEE Computer Press, Los Angeles (1996) 164–173
5. Blaze, M., Feigenbaum, J., Strauss, M.: Compliance Checking in the Policy Maker Trust Management System. *In Proceedings of the Financial Cryptography '98, Lecture Notes in Computer Science*, Vol. 1465 254–274
6. Brewer, D.F.C. and Nash, M.J.: The Chinese Wall Security Policy. *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages (1989) 206–214
7. Jajodia, S., Samarati, P. and Subrahmanian, V.S.: A logical Language for Expressing Authorizations. *Proceedings of the 1997 IEEE Symposium on Security and Privacy* (1997)
8. Lampson, B.: Protection. *ACM Operation System review 8(1)* (January 1974) 18–24
9. Lukasiewicz, J.: On Three-Valued Logic. 1920. *Ruch Filozoficzny* 1920, 5, pp.170-1. *English translation in Borkowski, L. (ed.) Jan Lukasiewicz: Selected Works.* Amsterdam: North Holland (1970)
10. Massimo, A., Cazzola, W., Fernandez, E.B.: A History-Dependent Access Control Mechanism Using Reflection *Proceedings of 5th ECOOP Workshop on Mobile Object Systems (EWMOS'99)*, (June 1999)
11. Moffet, J.D. and Sloman, M.S.: The representation of Policies as System objects. *Proceedings of the ACM Conference on Organizational Computing Systems*, Atlanta, GA (November 1991) 171–184

12. Neuman, B.C.: Proxy-based authorization and accounting for distributed systems. *Proceedings of the 13th International Conference on Distributed Computing Systems, Pittsburgh* (May 1993)
13. Prior, A.N.: Three-Valued Logic and Future Contingents. *Philosophical Quarterly.* Vol. 3 (1953) 17–26
14. Ryutov, T.V. and Neuman, B.C.: Representation and Evaluation of Security policies for Distributed system Services. *In Proceedings of the DARPA Information Survivability Conference and Exposition.* Hilton Head, South Carolina (January 2000)
15. Woo, T.Y.C. and Lam, S.S.: Authorization in distributed systems: a new approach. *Journal of Computer Security, 2* (1993) 107–136
16. Schneider, F.B.: Enforceable security policies. *Technical report TR98 1664*, Cornell University (January 1998)
17. Shen, W. and Dewan, P.: Access Control for Collaborative Environments. *Proceedings of CSCW* (November, 1992) 51–58
18. Simon, R.T. and Zurko, M.E.: Separation of Duty in Role-Based Environments *Computer Security Foundations Workshop* (June 1997)

Appendix F:

Representation and Evaluation of Security Policies for Distributed System Services

Tatyana Ryutov and Clifford Neuman
Information Sciences Institute
University of Southern California
4676 Admiralty Way suite 1001
Marina del Rey, CA 90292
{tryutov, bcn}@isi.edu
(310)822-1511 (voice) (310)823-6714 (fax)

Abstract

We present a new model for authorization that integrates both local and distributed access control policies and that is extensible across applications and administrative domains. We introduce a general mechanism that is capable of implementing several security policies including role-based access control, Clark-Wilson, ACLs, capabilities, and lattice-based access controls. The Generic Authorization and Access-control API (GAA API) provides a generic framework by which applications facilitate access control decisions and request authorization information about a particular resource. We have integrated our system with the Prospero Resource Manager and Globus Security Infrastructure.

1 Introduction

The conventional concept of an Access Control List (ACL) is the architectural foundation of many authorization mechanisms. A typical ACL is associated with an object to be protected and enumerates the list of authorized users and their rights to access the object. Access rights are selected from a predefined fixed set built into the authorization mechanism. Specification of the subjects is bound to the particular security mechanism employed by the system. The limitations of the traditional access control model become apparent when it is applied in a heterogeneous, administratively decentralized, distributed environment.

The variety of services available on the Internet continues to increase and new classes of applications are evolving, including metacomputing, remote printing, and video conferencing. These applications will require interactions between entities in autonomous security domains. The generic traditional access rights may not be sufficient for some applications to express authorization requirements. For example, a site might be willing to make its resources available to others, but limited to maximum CPU and memory utilization or based on a requirement for payment. It is difficult to specify such security policies in terms of conventional ACLs.

Specification of security policies for principals from multiple administrative domains poses additional problems:

- In a multipolicy environment, policy integration should incorporate the diverse authorization models that can coexist in a distributed system.
- The implementation will require integration of different sets of policies associated with the domain providing resources, the domain requesting resources and the individual users within each domain.
- There are multiple mechanisms for authentication of users in different domains. Therefore, there may be no single syntax for specification of principals.
- Administrators of each domain might use domain-specific policy syntax and heterogeneous implementations of the policies. Generalizing the way that applications define their security requirements provides the means for integration and translation of security policies across multiple authorization models.

This paper describes an authorization framework designed to meet these needs. Our framework is applicable for a wide range of systems and applications.

It includes a flexible mechanism for security policy representation and provides the integration of local and dis-

[0] In Proceedings of the DARPA Information Survivability Conference Exposition, January 2000. Hilton Head, South Carolina.

[0] 0-7695-0490-6/99 $10.00 ©1999 IEEE

tributed security policies. The system supports the common authorization requirements and provides the means for defining and integrating application or organization specific policies as well. We show how this mechanism can implement role-based access control, Clark-Wilson model, and lattice-based policies.

Our framework consists of two components, a policy language and the Generic Authorization and Access-control API.

- Policy language

  The language allows us to represent existing access control models (e.g. ACL, capability, lattice-based access controls) in a uniform and consistent manner. Authorization restrictions allow the administrator to define which operations are allowed, and under what conditions (e.g., user identity, group membership, or time of day). These restrictions may implement application-specific policies.

- Generic Authorization and Access-control API

  A common access control API facilitates the application integration of authentication and authorization. This API allows applications to request the authorization policy information for a particular resource and to evaluate this policy against credentials carried in the security context for the current connections. Applications invoke the GAA API functions to determine if a requested operation or set of operations was authorized or if additional checks are necessary.

2 Related Work

There has been recent work elsewhere on access control models for Internet user agents [7], [8]. These models apply to the Javakey utility as an authentication mechanism and use public key digital signatures. Our model is general enough to use a variety of security mechanisms based on public or secret key cryptosystems. Also, our model is application-independent whereas the models in [7] and [8] apply primarily for browser-like applications.

The Generalized Access Control List (GACL) framework described by Woo and Lam [3] presents a language-based approach for specifying authorization policies. The main goal of the GACL framework is merging policies associated with different objects and to resolve complex dependencies. GACL allows specification of the inheritance rules; access rights can be propagated from one object to the other. A gacl may reference other gacls in its entries. The benefit of the GACL approach is the ability to omit redundant information but it may require the retrieval and evaluation of more then one gacl. Specification of policy dependencies with inheritance is error-prone and may result in circular dependency of the policies and inconsistency may result.

More importantly, the expressive power of GACL is limited to that of ACL-based schemes and provides no support for capabilities and multi-level security systems. The GACL model supports only system state-related conditions within which rights are granted, such as current system load and maximum number of copies of a program to be run concurrently. This may not be sufficient for distributed applications. Our model allows fine-grained control over the conditions.

Policy management issues were addressed by Blaze, et. al. [9] with a claim that using PolicyMaker strengthens security. Because PolicyMaker credentials bind granted rights to public keys, instead of identities, this eliminates one level of indirection. Unfortunately, this binding complicates authorization management, and as applied in cases where a system uses X.509 or PGP certificates, this binding is dependent on the application which translates credentials to the PolicyMaker format.

Policies in the PolicyMaker format are easily expressed in our framework. We treat security policies as a set of operations that subjects are allowed to perform on the targeted objects, and optional constraints are placed on the granted operations. The basic question of access control is whether a subject is allowed to perform a requested operation. The GAA API provides a common interface for asking this question. In contrast, to use PolicyMaker an application developer must define an application-specific language describing the requested operation. This language might not be reusable across different application domains.

The related work described so far presents static policy evaluation mechanisms. Decisions are based on a set of policies and credentials presented at the time of the request. In contrast, our framework allows dynamic policy evaluation where credentials can be requested from the client or from third parties during recursive evaluation of policies within the API.

3 Overview of the Framework

Our framework is applied to distributed systems that span multiple autonomous administrative domains without a central management authority. Applications may impose their own security policies and use different authentication services, e.g. Kerberos, DCE or X.509 certificates. We assume that within a distributed system, multiple independent applications coexist.

The individual security requirements of each application are reflected in application-specific security policies. There might exist common ACLs that apply to sets of applications. Therefore, we designed a flexible and expressive mechanism for representing and evaluating authorization policies.

It is general enough to support a variety of security mechanisms based on public or secret key cryptosystems, and it is usable by multiple applications supporting different operations and different kinds of protected objects.

The major components of the architecture are:

- Authentication mechanisms perform authentication of users and supply credentials.
- A group server maintains group membership information.
- The GAA API; Applications call GAA API routines to check authorization against an authorization model. The API routines obtain policies from local files, distributed authorization servers, and from credentials provided by the user. They combine local and distributed authorization information under a single API based on the requirements of the application and applicable policies.
- Delegation is supported by delegation credentials, such as *restricted proxies* [1], or through other delegation methods.

3.1 Policy Language

The security policy associated with a protected resource consists of a set of allowed operations, a set of approved principals, and optional operation constraints. For example, a system administrator can define the following security policy to govern access to a printer: "Joe Smith and members of Department1 are allowed to print documents Monday through Friday, from 9:00AM to 6:00PM". This policy can be described by an ACL mechanism, where for each resource, a list of valid entities is granted a set of access rights. The same policy can be implemented using a capability mechanism. However, to do so, traditional ACL and capability abstractions must be extended to allow conditional restrictions on access rights. Therefore, in implementing a policy, it should be possible to define:
1) access identity
2) grantor identity
3) a set of access rights
4) a set of conditions
The policy language represents a sequence of tokens. Each token consists of:

- `Token Type`

  Defines the type of the token. Tokens of the same type have the same authorization semantics.

- `Defining Authority`

  Indicates the authority responsible for defining the value within the token type.

- `Value`

  The value of the token. Its syntax and semantics are determined by the token type. The name space for the value is defined by the `Defining Authority` field.

The rest of this section describes the user-level representation of the policy language tokens, which can be used to implement both ACLs and capabilities. More precise syntax is given in the Appendix.

3.1.1 Specification of Access Identity

The access identity represents an identity to be used for access control purposes. The authorization framework supports the following types of access identity: `USER`, `HOST`, `APPLICATION`, `CA` (Certification Authority), `GROUP` and `ANYBODY`. Where `ANYBODY` represents any entity regardless of authentication. This may be useful for setting the default policies. The type of access identity is useful in determining which additional credentials are needed (see section 3.3). Principals can be aggregated into a single entry when the same set of access rights and conditions applies to all of them.

Our framework supports multiple existing principal naming methods. Different administrative domains might use different authentication mechanisms, each having a particular syntax for specification of principals. Therefore, `Defining Authority` for access identity indicates the underlying authentication mechanism used to provide the principal identity. Value represents the particular principal identity.

3.1.2 Specification of Grantor Identity

The grantor identity represents an identity used to specify the grantor of a capability or a delegated credential. Its structure is similar to the one of the access identity described in the previous subsection.

3.1.3 Specification of Access Rights

It must be possible to specify which principals or groups of principals are authorized for specific operations, as well as who is explicitly denied authorizations, therefore we define positive and negative access rights.

All operations defined on the object are grouped by type of access to the object they represent, and named using a tag. For example, the following operations are defined for a file:
`Token Type:` *pos_access_rights*
`Defining Authority:` *local_manager*
`Value:` *FILE:read,write,execute*

However, in a bank application, an object might be a customer account, and the following set of operations might be defined:

`Token Type:` *pos_access_rights*
`Defining Authority:` *local_manager*
`Value:` *ACCOUNT:deposit,withdraw,transfer*

3.1.4 Specification of Conditions

Conditions specify the type-specific policies under which an operation can be performed on an object. A condition is interpreted according to its type. Conditions can be categorized as generic or specific. Generic conditions are evaluated within the access control API; specific conditions are application-dependent and usually are evaluated by the application. These are several of the more useful generic conditions [1].

- **time**

  Time periods for which access is granted.

- **location**

  Location of the principal. Authorization is granted to the principals residing on specific hosts, domains, or networks.

- **message protection**

  Required confidentiality/integrity message protection. This condition specifies a level or mechanism that must be used for confidentiality or integrity if access is to be granted.

- **privilege constraints**

  Specifies well-formed transactions and separation of duty constraints. For more details see Section 8.

- **multi-level security constraints**

  Specifies mandatory confidentiality and integrity constraints. For more information see Section 9.

- **payment**

  Specifies a currency and an amount that must be paid prior to accessing an object.

- **quota**

  Specifies a currency and a limit. It limits the quantity of a resource that can be consumed or obtained.

- **strength of authentication**

  Specifies the authentication mechanism or set of suitable mechanisms, for authentication.

- **trust constraints**

  Specifies restrictions placed on security credentials. For more information see Section 6.

- **attributes of subjects**

  Defines a set of attributes that must be possessed by subjects in order to get access to the object, e.g. security label.

If generic conditions are not sufficient for expressing application-specific security policies, applications specify their own conditions. Anything that can be expressed as an alphanumeric string can be a condition. The application must provide evaluation rules for the application-specific conditions, or be prepared to evaluate the condition once the authorization call completes.

3.1.5 Extended Access Control Lists (EACLs)

Extended Access Control Lists (EACLs) extend the conventional ACL concept by allowing one to specify conditional authorization policies. These are implemented as conditions on authentication and authorization credentials. An EACL is associated with an object and lists the subjects allowed to access this object and the type of granted access. For example, the following EACL implements policy stating that anyone authenticated by Kerberos.V5 has read access to the targeted resource and any member of group 15 connecting from the `USC.EDU` domain has read and write access to the object.

`Token Type:` *access_id_ANYBODY*
`Defining Authority:` *none*
`Value:` *none*

`Token Type:` *pos_access_rights*
`Defining Authority:` *local_manager*
`Value:` *FILE:read*

`Token Type:` *authentication_mechanism*
`Defining Authority:` *system_manager*
`Value:` *kerberos.V5*

`Token Type:` *access_id_GROUP*
`Defining Authority:` *DCE*
`Value:` *15*

`Token Type:` *pos_access_rights*
`Defining Authority:` *local_manager*
`Value:` *FILE:read FILE:write*

`Token Type:` *location*
`Defining Authority:` *system_manager*
`Value:` **.USC.EDU*

The framework supports various strengths of user authentication. A user may be granted a different set of rights, depending on the strength of the authentication method used for identification. Specification of weaker authentication methods including network address or username will allow the GAA API to be used with existing applications that do not have support for strong authentication.

Objects that need to be protected include files, directories, network connections, hosts, and auxiliary devices, e.g. printers and faxes. Our authorization mechanism supports these different kinds of objects in a uniform manner. The same EACL structure can be used to specify access policies for different kinds of objects. Object names are drawn from the application-specific name space and are opaque to the authorization mechanism.

When a protected object is created, an EACL is associated with the object. The management of EACLs, including giving authority to modify an EACL, is supported through inclusion of entries specifying which principals are allowed to modify the EACL. The control permissions comprise a separate set of access rights named with the tag *MANAGEMENT.* To restrict the ability to pass the control permissions to others a condition *no_delegation* may be specified associated with such entries.

3.1.6 Capabilities

Here we present an implementation of a capability. The example states that the capability granted by the group *admin* permits read access if the capability is presented during the specified time period.

```
Token Type: grantor_id_GROUP
Defining Authority: kerberos.V5
Value: admin@USC.EDU

Token Type: pos_access_rights
Defining Authority: local_manager
Value: FILE:read

Token Type: time_window
Defining Authority: eastern_timezone
Value: 8:00AM-5:00PM
```

3.2 EACL evaluation

The policy language we presented supports authorization models based on the closed world model, when all rights are implicitly denied. Authorizations are granted by an explicit listing of positive access rights. Restrictions placed on positive access rights have the goal of restricting the granted rights. The meaning of conditions on negative (denied) access rights is unclear. We intend to investigate this issue, however, for the time being, we require that:

1) A single EACL entry must not specify both positive and negative rights.

2) If an EACL entry specifies negative rights, it must not have any conditions. If both negative and positive authorizations are allowed in individual or group entries, inconsistencies must be resolved according to resolution rules. The design approach we adopted allows the ordered interpretation [11] of EACLs. Evaluation of ordered EACL starts from the first to the last in the list of EACL entries. The resolution of inconsistent authorization is based on ordering. The authorizations that already have been examined take precedence over new authorizations. Other interpretations were possible, but we found that for many such policies, resolution of inconsistencies was either NP-Complete or undecidable.

There may be interactions when independent credentials are used, e.g., one set of credentials causes denial, but the other causes accept. A user may chose to withhold credentials that it believes may result in a denial. The administrator must deal with these issues by carefully setting policies in an EACL. Conflicts may arise when more then one entry applies. For example, one matching entry specifies individual subject (user, host or application), and another matching entry specifies a certain group name. In this case, we would require the entry for the individual subject to be placed before the entry for the group (assuming the policy expressed for the individual subject entry is an exception to the policy expressed for the group entry). When several EACL entries with different conditions apply, entries for which conditions are not satisfied will not affect the outcome of the authorization function.

An ordered evaluation approach is easier to implement as it allows only partial evaluation of an EACL and resolves the authorization conflicts. The problem with this approach is that it requires total ordering among authorizations. It requires careful writing of the EACL by the security administrator and is error-prone. An improper order of the EACL entries may result in discrepancies between the intended policy and the one that results from evaluation of the EACL. It might be useful to have a separate module [4], [9], that would help verify and debug the EACL to assure that it expresses the desired policy.

3.3 Credential evaluation

Credentials are translated to the GAA API internal format and placed into the GAA API security context. When evaluating an EACL, the security context is searched for the necessary credentials. Assume that file *doc.txt* has the following EACL shown in **Table 1.** stored in the authorization data base:

		IDENTITY	ACCESS RIGHTS
	TOKEN TYPE	access_id_USER	pos_access_rights
#1	DEF. AUTHORITY	KerberosV5	local_manager
	VALUE	tom@ORG.EDU	FILE : read

		IDENTITY	ACCESS RIGHTS
	TOKEN TYPE	access_id_GROUP	pos_access_rights
#2	DEF. AUTHORITY	KerberosV5	local_manager
	VALUE	admin@ORG.EDU	FILE : read,write

		IDENTITY	ACCESS RIGHTS
	TOKEN TYPE	access_id_USER	pos_access_rights
#3	DEF. AUTHORITY	KerberosV5	local_manager
	VALUE	joe@ORG.EDU	FILE : write

Table 1.

Credentials may have optional conditions associated with the granted rights. Assume the following credentials are stored in the security context associated with the user Tom.

Identity credential:

access_id_USER kerberos.v5 tom@ORG.EDU
`condition:` *time_window pacific_tzone 6am-7pm*

Group membership credential:

access_id_GROUP kerberosV5 admin@ORG.EDU
`condition:` *privilege:restricted*

Delegation credential:

`grantor:` *grantor_id_USER kerberosV5 joe@ORG.EDU*
`grantee:` *access_id_USER kerberosV5 tom@ORG.EDU*
`objects:` *doc.txt*
`rights:` *pos_access_rights local_manager FILE:write*
`condition:` *location local_manager *.org.edu*

Let's consider a request from a user Tom who is connecting from the *ORG.EDU* domain to write to the file *doc.txt* at 5pm.

In evaluating the EACL, the first entry does not grant the requested operation, however the second entry grants it. The evaluation function will then check the security context for the group `admin` membership credential. The proper credential is found, however, there is a condition `privilege:restricted`. This means that Tom can use this privilege only if logged in as an administrator. Evaluation continues. The third entry grants the requested operation. The evaluation function will look for a delegation credential for `tom@ORG.EDU` issued by `joe@ORG.EDU`. The appropriate delegation credential is found. The condition on location `*org.edu` is satisfied, so the requested access will be granted.

3.4 Generic Authorization and Access-control API (GAA API)

In this section we provide a description of the main GAA API routines.

3.4.1 GAA API functions

The `gaa_get_object_policy_info` function is called to obtain the security policy associated with the object.

- **Input**:
  - *Reference to the object to be accessed.* The identifier for the object is from an application-dependent name space, it can be represented as unique object identifier, or symbolic name local to the application.
  - *Pointer to application specific Authorization Database*.
  - *Upcall function for the retrieval of the object policy*. The application maintains authorization information in a form understood by the application. It can be stored in a file, database, directory service or in some other way. The upcall function provided for the GAA API retrieves this information and translates it into the internal representation understood by the GAA API.
- **Output**:
  - *Object policy handle*

The `gaa_check_authorization` function tells the application server whether the requested operations are authorized, or if additional application-specific checks are required.

- **Input**:
  - *Object policy handle, returned by* `gaa_get_object_policy_info`
  - *Principal's security context* (see section 3.5.1)
  - *Operations for authorization*. This argument indicates requested operations.
- **Output**:
  - `YES` (indicating authorization) is returned if all requested operations are authorized.
  - `NO` (indicating denial of authorization) is returned if at least one operation is not authorized.

  - `MAYBE` (indicating a need for application-specific checks) is returned if there are some unevaluated conditions and additional application-specific checks are needed, or if continuous evaluation of conditions is required.
  - *detailed answer* contains:
    - ∗ Authorization valid time period. The time period during which the authorization is granted is returned as condition to be checked by the application.
      Expiration time is calculated by the GAA API, based on:
      1. Time-related conditions in the object policy, e.g. EACL matching entries.
      2. Restrictions in the authentication and authorization credentials.
    - ∗ The requested operations are returned marked as granted or denied along with a list of corresponding conditions, if any. Each condition is marked as evaluated or not evaluated, and if evaluated marked as met, not met or further evaluation or enforcement is required. This tells the application which policies must be enforced.
    - ∗ Information about additional security attributes required. Additional credentials might be required from clients to perform certain operations, e.g. group membership or delegated credentials.

- `gaa_inquire_object_policy_info`

  This function allows the application to discover access control policies associated with the targeted object applied to a particular principal. It returns a list of rights that the principal is authorized for and corresponding conditions, if any. The application must understand the conditions that are returned unevaluated, or it must reject the request. If understood, the application checks the conditions against information about the request, the target object, or environmental conditions to determine whether the conditions are met. Actual enforcement of policies expressed through application specific conditions is the responsibility of the application and is outside of the scope of this paper.

3.4.2 GAA API Security Context

The security context is a GAA API data structure. It stores information relevant to access control. Some of its constituents are listed here:

Identity Verified authentication information, such as principal ID for a particular security mechanism. To determine which entries apply, the GAA API checks if the specified principal ID appears in an EACL entry that is paired with a privilege for the type of access requested.

Authorization Attributes Verified authorization credentials, such as group membership, group non-membership, delegation credentials, and capabilities.

Evaluation and Retrieval Functions for Upcalls These functions are called to evaluate application-specific conditions, to request additional credentials, and to verify them.

4 Creation of the GAA API security context

Prior to calling the `gaa_check_authorization` function, the application must obtain the authenticated principal's identity and store it in the security context. This context may be constructed from credentials obtained from different mechanisms, e.g. GSS API, Kerberos, or others. This scenario places a heavy burden on the application programmer to provide the integration of the security mechanism with the application. A second scenario is to obtain the authentication credentials from a transport protocol that already has the security context integrated with it. For example, the application can call SSL or authenticated RPC. In this case, it is the implementation of the transport mechanism (usually written by someone other than the application programmer) which calls the security API requesting principal's identity.

The principal's authentication information is placed into the security context and passed to the GAA API. When additional security attributes are required for the requested operation, the list of required attributes is returned to the application, which may request them. Through the security context, the application may provide the GAA API with an upcall function for requesting required additional credentials. The credentials pulled by the GAA API are verified and added to the security context by the upcall function.

5 An Extended Example

To illustrate our approach we describe a simple Printer Manager application, where protected objects are printers. The Printer Manager accepts requests from users to access printers and invokes the GAA API routines to make authorization decisions, under the assumption that the administrator of the resources has specified the local policy regarding the use of the resources by means of EACL files. These files are stored in an authorization database, maintained by the Printer Manager.

5.1 Conditions

Administrators will be more willing to grant access to the printers if they can restrict the access to the resources to only users and organizations they trust. Further, the administrators may need to specify time availability, restrictions on resources consumed by the clients and accounting for the consumed resources. To specify these limits, the Printer Manager uses generic conditions, such as time, location, payment and quota. As an example of Printer Manager-specific condition, consider printer load, expressed as maximum number of jobs that may be in the queue.

5.2 Authorization Walk-through

Here we present an authorization scenario to demonstrate the use of the authorization framework for the case of printing a document. Assume Kerberos V5 is used for principal authentication. Assume that printer *ps12a* has the following ordered EACL shown in **Table 2.** stored in the Printer Manager authorization database.

		IDENTITY	ACCESS RIGHTS	CONDITIONS	
#1	TOKEN TYPE	access_id_USER	pos_access_rights	time_window	printer_load
	DEF. AUTHORITY	KerberosV5	local_manager	pacific_tzone	local_manager
	VALUE	joe@ORG.EDU	PRINTER : submit_print_job	6AM-8PM	20%

		IDENTITY	ACCESS RIGHTS	
#2	TOKEN TYPE	access_identity_GROUP	positive_access_rights	positive_access_rights
	DEF. AUTHORITY	KerberosV5	local_manager	local_manager
	VALUE	operator@ORG.EDU	PRINTER : *	DEVICE : power_down
	TOKEN TYPE	access_identity_USER		
	DEF. AUTHORITY	KerberosV5		
	VALUE	tom@ORG.EDU		

		IDENTITY	ACCESS RIGHTS	CONDITIONS	
#3	TOKEN TYPE	access_id_ANYBODY	pos_access_rights	time_day	time_window
	DEF. AUTHORITY	none	local_manager	local_manager	pacific_tzone
	VALUE	none	PRINTER:view_printer_capabilities	sat-sun	6AM-8PM

Table 2.

Let's consider a request from user Tom who is connecting from the `ORG.EDU` domain to print a document on the printer *ps12a* at 7:30 PM.

When a client process running on behalf of the user contacts the Printer Manager with the request to `submit_print_job` to printer *ps12a*, the Printer Manager first calls `gaa_get_object_policy_info` to obtain a handle to the EACL of printer *ps12a*. The upcall function for retrieving the EACL for the specified object from the Authorization Database system is passed to the GAA API and is called by `gaa_get_object_policy_info`, which returns the EACL handle.

The Printer Manager must place the principal's authenticated identity in the security context to pass into the `gaa_check_authorization` function. This context may be constructed according to the first or second scenario, described in Section 8. If Tom is authenticated successfully, then verified identity credentials are placed into the security context, specifying Tom as the Kerberos principal `tom@ORG.EDU`.

Next, the Printer Manager calls the `gaa_check_authorization` function. In evaluating the EACL, the first entry applies. It grants the requested operation, but there are two conditions that must be evaluated.

The first condition is generic and is evaluated directly by the GAA API. Since, the request was issued at 7:30 PM this condition is satisfied. The second condition is specific. If the security context defined a condition evaluation function for upcall, then this function is invoked and if this condition is met then the final answer is `YES` (authorized) and detailed answer contains an authorization expiration time : 8PM (assume that authentication credential has expiration time 9PM), allowed operation `submit_print_job` and two conditions. Both conditions are marked as evaluated and met. During the execution of the task the Printer Manager is enforcing the limits imposed on the local resources and authorization time.

If the corresponding upcall function was not passed to the GAA API, the answer is `MAYBE` and the second condition is marked as not evaluated and must be checked by the Printer Manager.

When additional credentials are needed, if the security context defines a credential retrieval function for the upcall, it is invoked. If the requested credential is obtained, then the final answer is `YES`. If the upcall function was not passed to the GAA API, the answer is `NO`.

6 Integration with alternative authentication mechanisms

Our model is designed for a system that spans multiple administrative domains where each domain can impose its own security policies. It is still necessary that a common authentication mechanism be supported between two communicating systems. The model we present enables the syntactic specification of multiple authentication policies and the unambiguous identification of principals in each, but it does not translate between heterogeneous authentication mechanisms.

We have integrated our distributed model for authorization with the Prospero Resource Manager (PRM), a metacomputing resource allocation system developed at USC. PRM uses Kerberos [2] to achieve strong authentication. PRM uses calls to the Asynchronous Reliable Delivery Protocol (ARDP) [16], a communication protocol which handles a set of security services, such as authentication, integrity and payment. ARDP calls the Kerberos library through a security API, requesting the principal's authentication information.

In addition, we have integrated the framework with the Globus Security Infrastructure (GSI), a component of the

Globus metacomputing Toolkit [18]. GSI is implemented on top of the GSS-API which allows the integration of different underlying security mechanisms. Currently, GSI implementation uses SSL authentication protocol with X.509 certificates.

Public key authentication requires consideration of the trustworthiness of the certifying authorities for the purpose of public key certification. Authentication is not based on the public key alone, since anybody can issue a valid certificate.

Certificates can comprise a chain, where each certificate (except the last one) is followed by a certificate of its issuer. Reliable authentication of a public key must be based on a complete chain of certificates which starts at an end-entity (e.g. user) certificate, includes zero or more Certification Authorities (CA) certificates and ends at a self-signed root certificate. A policy must be specified to validate the legitimacy of the received certificate chain and the authenticity of the specified keys. The following is an example of an EACL used for describing the Globus policy for what CAs are allowed to sign which certificates. The Globus CA can sign certificates for Globus or the Alliance. The Alliance CA can sign certificates for the Alliance.

`Token Type:` *access_id_CA*
`Defining Authority:` *X509*
`Value:` */C=US/O=Globus/CN=Globus CA*

`Token Type:` *pos_access_rights*
`Defining Authority:` *globus*
`Value:` *CA:sign*

`Token Type:` *cond_subjects*
`Defining Authority:` *globus*
`Value:` */C=us/O=Globus/* /C=us/O=Alliance/**

7 Groups and Roles

A group is a convenient method to associate a name with a set of subjects and to use this group name for access control purposes. The kind of subject (individual user, host, application or other group) composing the group is opaque to the authorization mechanism. A group server issues group membership and non-membership certificates.

In general, a principal may be a member of several groups. By default, a principal operates with the union of privileges of all groups to which it belongs, as well as all of his individual privileges.

Some applications adopt role-based access control. The concept of roles is not consistent across different systems. Several definitions of roles are present in the literature. In general, a role is named collection of privileges needed to perform specific tasks in the system. Role properties [4] include:

- A user can be a member of several roles
- Role can be activated and deactivated by users at their discretion.
- Authorizations given to a role are applicable only when that role is activated.
- There may be various constraints placed on the use of roles, e.g. a user can activate just one role at a time.

Shandu et. al. [10] view roles as a policy and groups as a mechanism for role implementation. We adopt this point of view. In our framework we implement different flavors of roles using the notion of group and a set of restrictions on granted privileges. Consider a role-based policy, which assigns users: Tom, Joe, and Ken role `Bank_Teller`. This role allows a legitimate user to perform deposit and withdraw operations on objects *account_1* and *account_2*. This policy may be easily expressed by our EACL framework:

1. Group `Bank_Teller` is defined which will include Tom, Joe, and Ken
2. The EACLs for objects *account_1* and *account_2* will contain the following entry:

   `Token Type:` *access_id_GROUP*
   `Defining Authority:` *X.509*
   `Value:` */C=US/O=Globus/CN=Bank Teller*

   `Token Type:` *pos_access_rights*
   `Defining Authority:` *pasific_coast_bank*
   `Value:` *ACCOUNT:deposit,withdraw*

In expressing role-based policy using groups, the issue of constraints on role activation and use should be addressed.

8 Clark-Wilson

The Clark-Wilson model [12] was developed to address security issues in commercial environments. The model uses two categories of mechanisms to realize integrity: well-formed transactions and separation of duty.

Our framework is designed to handle the Clark-Wilson integrity model. A possible way to represent a constraint that only certain trusted programs can modify objects is using `application:checksum` condition, where the checksum ensures authenticity of the application. Another way is using `application:endorser` condition, which indicates that a valid certificate, stating that the application has been endorsed by the specified endorser, must be presented.

Static separation of duty is enforced by the security administrator when assigning group membership. Dynamic separation of duty enforces control over how permissions are used at the access time [6]. Here are examples of EACL conditions specific to the Dynamic separation of duty:

- `privilege:restricted` Makes subject operate with the privilege of only one group at a time.
- `privilege:set_of_groups` Makes subject operate with the privilege of only specified groups at a time.
- `endorsement:list_of_endorsers` Concurrence of several subjects to perform some operation.

9 Lattice-based Policies

Our framework allows incorporation of Mandatory Confidentiality [14], Mandatory Integrity [15] models and their combination.

Mandatory policies govern access on the basis of classification of subjects and objects in the system. Objects and subjects are assigned security labels:

1. Confidentiality labels, e.g. Top_Secret/NASA, Sensitive/Department2
2. Integrity labels, e.g. High_integrity, Low_integrity
3. Single security labels for both confidentiality and integrity, e.g. Top_Secret/NASA, Unclassified. Assume that the first label denotes high integrity level, whereas the second one denotes low integrity level.

To prove eligibility to access an object, a subject has to present a valid credential, stating subject's security label.

All access rights are divided into read-class and write-class. Appropriate rules are applied to each class.

Generic conditions for read-class access rights:

a) `conf_read_equal:cofidentiality_label`

This condition specifies that a subject, wishing to get read-class access to the object has to have security clearance equal to the one, specified in the cofidentiality_label field.

b) `conf_read_below:cofidentiality_label`

This condition is used to enforce `read down` mandatory confidentiality rule. It specifies that a subject, wishing to get read-class access to the object has to have security clearance no less the one, specified in the `cofidentiality_label` field.

c) `integr_read_equal:integrity_label`

This condition specifies that a subject, wishing to get read-class access to the object has to have security clearance equal to the one, specified in the `integrity_label` field.

d) `integr_read_above:integrity_label`

This condition is used to enforce `read up` mandatory integrity rule. It specifies that a subject, wishing to get read-class access to the object has to have integrity clearance less or equal to the one, specified in the `integrity_label` field.

Similarly we define generic conditions for write-class access rights. Assume file *doc.txt* has classification Sensitive/Departmen1 and integrity label Medium, then EACL for this file can be specified as:

		IDENTITY	ACCESS RIGHTS	CONDITIONS	
#1	TOKEN TYPE	access_id_ANIBODY	pos_access_rights	conf_write_above	integr_write_below
	DEF. AUTHORITY	none	system_manager	system_manager	system_manager
	VALUE	none	FILE : write	Sensitive/Deprt1	Medium
			pos_access_rights	conf_read_below	
			system_manager	system_manager	
			FILE : read	Sensitive/Deprt1	

Table 3.

Note that in the example above, everybody in the distributed system can get read or write access to the file if a valid credential stating the appropriate security label attribute is presented. This poses a requirement that security labels be unique across different security domains. This may not be easily satisfied.

A possible way to restrict the scope of security labels to a particular administrative domain is to specify an additional condition such as location.

10 Conclusions

In this paper we presented a generic authorization mechanism that supports a variety of security mechanisms based on public or secret key cryptography. The mechanism is extensible across multiple applications supporting different operations and different kinds of protected objects. Alternative implementations may be chosen for underling security services that support the API. By extending the traditional ACLs and capabilities with conditions on authorized rights we are able to support a flexible distributed authorization mechanism, allowing applications and users to define their own access control policies either independently or in conjunction with centralized authorization and group servers. The problem of policy translation is addressed by using generic or application-specific evaluation functions. We are going to investigate the request and evaluation of additional credentials. The assumption that all relevant credentials are passed for evaluation contradicts privacy requirements. It might not be always desirable to reveal group membership and principal attributes up front. We have integrated our model with several applications.

11 Appendix

We use the Backus-Naur Form to denote the elements of our policy language. Square brackets, `[ ]`, denote optional items and curly brackets, {}, surround items that can repeat zero or more times. A vertical line, **|**, separates alternatives. Items inside double quotes are the terminal symbols.

An EACL is specified according to the following format:

eacl ::= {eacl_entry}

eacl_entry ::=
access_id {access_id} pos_access_rights {condition}
{pos_access_rights {condition}} **|**
access_id {access_id} neg_access_rights

access_id ::=
access_id_type def_authority value

access_id_type ::=
”access_id_HOST” **|**
”access_id_USER” **|**
”access_id_GROUP” **|**
”access_id_CA” **|**
”access_id_APPLICATION” **|**
”access_id_ANYBODY”

A capability is defined according to the following format:

capability ::=
grantor_id pos_access_rights {condition}
{pos_access_rights {condiction}}

grantor_id ::=
grantor_id_type def_authority value

grantor_id_type ::=
”grantor_id_HOST” **|**
”grantor_id_USER” **|**
”grantor_id_GROUP” **|**
”grantor_id_CA” **|**
”grantor_id_APPLICATION” **|**
”grantor_id_ANYBODY”

pos_access_rights ::=
”pos_access_rights” def_authority value
{”pos_access_rights” def_authority value}

neg_access_rights ::=
”neg_access_rights” def_authority value
{”neg_access_rights” def_authority value}

condition ::=
condition_type def_authority value

condition_type ::= alphanumeric_string

def_authority ::= alphanumeric_string

value ::= alphanumeric_string

12 Acknowledgments

This research was supported in part by the Information Technology Office of the Defense Advanced Research Projects Agency (DoD) under the Scalable Computing Infrastructure (SCOPE) Project, Contract No. DABT63-95-C-0095, Security Infrastructure for Large Distributed Systems (SILDS) Project, Contract No. DABT63-94-C-0034, Global Operating Systems Technology (GOST) Project, Contract No. J-FBI-95-204, Diplomat, Project Department of Energy Cooperative Agreement No. DE-FC03-99ER25397 and by a grant from Xerox Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Intelligence Center and Fort Huachuca Directorate of Contracting, the Defense Advanced Research Projects Agency, the U.S. Government, or Xerox Corporation.

References

[1] C. Neuman. Proxy-based authorization and accounting for distributed systems. *Proceedings of the 13th International Conference on Distributed Computing Systems, Pittsburgh*, May 1993.

[2] C. Neuman and T. Ts’o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, pages 33–38, September 1994.

[3] T. Y. C. Woo and S.S. Lam. Designing a Distributed Authorization Service. *In Procedings IEEE INFOCOM ’98*, San Francisco, March 1998.

[4] S. Jajodia, P. Samarati and V.S. Subrahmanian. A logical Language for Expressing Authorizations. *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, 1997.

[5] M. Abadi, M. Burrows, B. Lampson and G. Plotkin A calculus for Access Control in Distributed Systems. *ACM Transactions on Programming Languages and Systems, Vol. 15, No 4,* Pages 706-734, September 1993.

[6] R. T. Simon and M. E. Zurko Separation of Duty in Role-Based Environments. *Computer Security Foundations Workshop*, June 1997.

[7] N. Nagaratnam and S. B. Byrne. Resource access control for internet user agent. *Proceedings of the third USENIX Conference on Object-Oriented Technologies and Systems, Portland, Oregon*, June 1997.

[8] L. Gong and R. Schemers. Implementing Protection Domains in the Java Development Kit 1.2. *Proceedings of Network and Distributed System Security Symposium, San Diego, California*, March 1998.

[9] M. Blaze, J. Feigenbaum and J. Lacy. Decentralized Trust Management. in *Proc. IEEE Symp. on Security and Privacy, IEEE Computer Press, Los Angeles*, pages 164-173, 1996.

[10] R. S. Shandhu, E. J. Coyne, et al Role-Based Access Control: A Multi-Dimensional View. *Proc. of 10th Annual Computer Security Applications Conference*, December 5-9, pages 54-62, 1994.

[11] W. Shen and P. Dewan Access Control for Collaborative Environments. *Proc. of CSCW*, November, 1992, pages 51-58

[12] D. D. Clark and D. R. Wilson Non Discretionary Controls Commercial Applications. *Proc. of the IEEE Symposium on Security and Privacy*, pages 184-194, April 1997.

[13] S B. Lipner A Comparison of Commercial and Military Computer Security Policies *Proc. of the 1987 IEEE Symposium on Security and Privacy*, 1982.

[14] D. Elliott Bell and L. J. LaPadula *Secure Computer System: Unified Exposition and Multics. Interpretation, ESD-TR-75-306 (MTR-2997), The MITRE Corporation* Bedford, Massachusetts, July 1975.

[15] K. J. Biba Integrity Considerations for Secure Computer Systems, *The MITRE Corporation*, Bedford, MA, MTR-3153, 30 June 1975.

[16] N. Salehi, K. Obraczka and C. Neuman The performance of a reliable, request-response transport protocol. *Proceedings of the Fourth IEEE Symposium on Computers and Communications*, 6-8 July, 1999.

[17] Edited by I. Foster and C. Kesselman.
The GRID: Blueprint for a New Computing Infrastructure *Morgan Kauffman Publishers*, 1999.

[18] I. Foster and C. Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kauffman Publishers, 1999.